# MICROSOFT®

*The High Performance Software*™

Library Reference

Library Reference

# Microsoft® C

## Run-Time Library Reference

## for the MS™-DOS Operating System

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1984, 1985

If you have comments about the software or this manual, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Microsoft, the Microsoft logo, and XENIX are registered trademarks, and MS is a trademark of Microsoft Corporation.

UNIX is a trademark of Bell Laboratories.

Document Number 8417L-300-00
Part Number 048-014-026

# Contents


## Part 1 Overview 1

### 1 Introduction 3

1.1 About the C Library 5
1.2 About This Manual 6
1.3 Notational Conventions 8

### 2 Using C Library Routines 11

2.1 Introduction 13
2.2 Identifying Functions and Macros 13
2.3 Including Files 15
2.4 Declaring Functions 16
2.5 Argument Type-Checking 17
2.6 Error Handling 19
2.7 Filenames and Pathnames 20
2.8 Binary and Text Modes 22
2.9 MS-DOS Considerations 24
2.10 Floating-Point Support 25

### 3 Global Variables and Standard Types 27

3.1 Introduction 29
3.2 daylight, timezone, tzname 29
3.3 _doserrno, errno, sys_errlist, sys_nerr 30
3.4 _fmode 31
3.5 _osmajor, _osminor 31
3.6 environ, _psp 32
3.7 Standard Types 33

# 4 Run-Time Routines by Category 35

4.1 Introduction 37
4.2 Buffer Manipulation 37
4.3 Character Classification and Conversion 38
4.4 Data Conversion 40
4.5 Directory Control 40
4.6 File Handling 41
4.7 Input and Output 42
4.8 Math 54
4.9 Memory Allocation 55
4.10 MS-DOS Interface 57
4.11 Process Control 59
4.12 Searching and Sorting 61
4.13 String Manipulation 62
4.14 Time 63
4.15 Miscellaneous 64

# 5 Include Files 67

5.1 Introduction 69
5.2 assert.h 70
5.3 conio.h 70
5.4 ctype.h 70
5.5 direct.h 71
5.6 dos.h 71
5.7 errno.h 72
5.8 fcntl.h 72
5.9 io.h 73
5.10 malloc.h 73
5.11 math.h 73
5.12 memory.h 74
5.13 process.h 74
5.14 search.h 75
5.15 setjmp.h 75
5.16 share.h 75
5.17 signal.h 75
5.18 stdio.h 76
5.19 stdlib.h 77
5.20 string.h 78

5.21 syslocking.h 78
5.22 sysstat.h 78
5.23 systimeb.h 79
5.24 systypes.h 79
5.25 sysutime.h 79
5.26 time.h 79
5.27 v2tov3.h 80

**Part 2 Reference 81**

**Appendices 369**

**A Error Messages 371**

A.1 Introduction 373
A.2 errno Values 374
A.3 Math Errors 376

**B A Common Library for XENIX and MS-DOS 377**

B.1 Introduction 379
B.2 Common Run-Time Routines 379
B.3 Common Global Variables 381
B.4 Common Include Files 383
B.5 Differences Between Common Routines 384

**Index 395**

# Tables

Table 4.1 Forms of the spawn and exec Routines 61

Library Reference

# Overview

# Part 1
# Overview


1 Introduction 3

2 Using C Library Routines 11

3 Global Variables and Standard Types 27

4 Run-Time Routines by Category 35

5 Include Files 67

# Chapter 1
# Introduction

1.1 About the C Library 5
1.2 About This Manual 6
1.3 Notational Conventions 8

## 1.1 About the C Library

The Microsoft® C Run-Time Library is a set of over 200 predefined functions and macros designed for use in C programs. The run-time library makes programming easier by providing:

1. An interface to operating system functions (such as opening and closing files).
2. Fast and efficient routines to perform common programming tasks (such as string manipulation), sparing the programmer the time and effort needed to write such functions.

The run-time library is especially important in C programming because C programmers rely on the library for basic functions not provided by the language. These functions include, among others, input and output, storage allocation, and process control.

The routines in the Microsoft C Run-Time Library have been designed to maintain maximum compatibility between MS™-DOS and XENIX® or UNIX™ systems. Throughout this manual, references to XENIX systems are intended to encompass UNIX and UNIX-like systems as well.

Most of the routines in the C run-time library for MS-DOS operate compatibly with routines having the same names in the C run-time library for XENIX operating systems. If you are interested in portability, see Appendix B, "A Common Library for XENIX and MS-DOS." This appendix lists the routines of the run-time library that are specific to MS-DOS and describes differences (if any) between the operation of routines with the same names on XENIX and MS-DOS.

For additional compatibility, the math routines of the Microsoft C Run-Time Library have been extended to provide exception handling in the same manner as UNIX System V math routines.

For programmers interested in taking advantage of the specific features of MS-DOS, the library includes MS-DOS interface functions. These functions allow MS-DOS system calls and interrupts to be invoked from a C program. The library also contains console input and output routines to allow efficient reading and writing from the user's console.

To take advantage of the Microsoft C Compiler's type-checking capabilities, the include files that accompany the run-time library have been expanded. In addition to the definitions and declarations required by library functions and macros, the include files now contain function declarations with argument type lists. The argument type lists enable type-checking for calls to library routines. This feature can be extremely helpful in detecting subtle program errors resulting from type mismatches between actual and formal arguments to a function, and its use is highly recommended. However, you are not required to use argument type-checking. The function declarations in the include files are enclosed in preprocessor **#ifdef** blocks, and are enabled only when you define the identifier **LINT_ARGS**.

To provide argument type lists for all run-time functions, several new include files have been added to the list of standard include files for the C run-time library. The names of the new include files have been chosen to maintain as much compatibility as possible with the proposed ANSI (American National Standards Institute) standard for C and with XENIX and UNIX names.

## 1.2 About This Manual

The *Microsoft C Run-Time Library Reference* describes the contents of the Microsoft C Run-Time Library. The manual assumes that you are familiar with the C language and with MS-DOS. It also assumes that you know how to compile and link C programs on your MS-DOS system and that you can set up a compiler and linker environment using environment variables. If you have questions about compiling, linking, or setting up an environment, see the *Microsoft C Compiler User's Guide,* which covers these topics. If you have questions about the C language, turn to the *Microsoft C Language Reference.*

The *Microsoft C Run-Time Library Reference* has two major parts. Part 1, "Overview," gives an introduction to the C run-time library. It discusses general rules that apply to the run-time library as a whole and summarizes the elements of the run-time library.

Part 2, "Reference," gives descriptions of the run-time routines in alphabetical order for quick reference. Once you have familiarized yourself with the library rules and procedures, you will probably use the second part of the manual most often.

The remaining chapters of Part 1 are as follows.

Chapter 2, "Using C Library Routines," gives general rules for understanding and using C library routines and mentions special considerations that apply to certain routines. It is recommended that you read this chapter before using the run-time library; you may also want to turn back to Chapter 2 when you have questions about library procedures.

Chapter 3, "Global Variables and Standard Types," describes variables and types that are defined in the run-time library and used by run-time library routines. This chapter also provides a cross-reference to the include file that defines or declares each variable or type. You may find these variables and types useful in your own functions. The variables and types are also described on the reference pages for the routines that use them in Part 2, "Reference."

Chapter 4, "Run-Time Routines by Category," breaks down the run-time library routines by category, lists the routines that fall into each category, and discusses considerations that apply to each category as a whole. The chapter is intended to complement Part 2, "Reference," making it easy to locate routines by task. Once you have located the name of the routine you want, you will need to turn to the appropriate page in Part 2, "Reference," for a detailed description.

Chapter 5, "Include Files," summarizes the contents of each include file provided with the run-time library.

The appendices, which follow Part 2, provide more detailed information about error messages and about XENIX-compatible routines. Appendix A, "Error Messages," describes the error values and messages that can appear when using library routines. Appendix B, "A Common Library for XENIX and MS-DOS," lists routines of the MS-DOS C library that operate compatibly with routines of the same name on XENIX (and UNIX) systems; Appendix B also describes any differences between the MS-DOS and XENIX versions of the routines. Common global variables and include files are also discussed in this appendix.

The remainder of this chapter describes the notational conventions used throughout the manual.

## 1.3 Notational Conventions

The following notational conventions are used throughout this manual.

*italics*

Italics are used for the names of arguments to library routines. In an actual program, a specific name or value replaces the italicized argument name. For example, in

**double atof**(*string*);

the argument *string* is italicized to indicate that this is the general form for the **atof** routine. In an actual program, the user supplies a particular argument for the placeholder *string*.

Italics are also used when referring to specific identifiers supplied in program examples for variables and functions. For instance, when a program example such as

```
y = floor(x);
```

is provided, the variable names *x* and *y* are italicized in the discussion of the example.

The names of library include files (for example, *stdio.h*) appear in italics when discussed in the text.

Occasionally, italics are used to emphasize particular words in the text.

[brackets]

Brackets enclose optional arguments in the specification for each library routine. However, the C language also uses brackets for array declarations and subscript expressions. In discussions of the syntax for array declarations and

subscript expressions and in examples, brackets have the meaning specified by C. For example, in

**int open**(*pathname, oflag* [, *pmode*]);

the brackets around ", *pmode*" indicate that the third argument (*pmode*) is optional and that, when given, *pmode* must be separated from the previous argument with a comma.

On the other hand,

```
char *args[4];
```

is an example showing the declaration of a 4-element array.

ellipses
.
.
.

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, two statements are shown. The ellipses between the statements indicate that intervening program lines occur but are not shown.

```
int x, y;
      .
      .
      .
y = abs(x);
```

CAPITALS

Capital letters are used for the names of environment variables (such as TZ and PATH) and MS-DOS commands (such as SET and PATH). However, on MS-DOS, you are not required to use capital letters for these variables and commands.

SMALL CAPITALS

Small capital letters are used for the names of keys and key sequences such as CONTROL-C.

| | |
|---|---|
| "quotation marks" | Quotation marks are used to set off program fragments or terms defined within the body of the text.<br><br>Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than " ". For example,<br><br>`"abc"`<br><br>is a C string. |
| **keywords** and **names** of library items | C keywords, such as **double** and **char**, are set in a different type font (the Helvetica font) to distinguish them from ordinary identifiers and text.<br><br>The names of run-time library routines, global variables, standard types, constants, and identifiers used by the C library are also set in this font to emphasize that these names are reserved by the run-time library. For example, the routine name **strcpy** appears in this font; so does the manifest constant **NULL**. |
| `programming examples` | Programming examples are displayed without proportional spacing so that they look similar to the programs you create with a text editor. |

# Chapter 2
# Using C Library Routines


2.1 Introduction 13
2.2 Identifying Functions and Macros 13
2.3 Including Files 15
2.4 Declaring Functions 16
2.5 Argument Type-Checking 17
2.6 Error Handling 19
2.7 Filenames and Pathnames 20
2.8 Binary and Text Modes 22
2.9 MS-DOS Considerations 24
2.10 Floating-Point Support 25

## 2.1 Introduction

To use a C library routine, you simply call it in your program, just as if the function were defined in your program. The C library functions are stored in compiled form in the library files that accompany your C compiler software.

At link time, your program must be linked with the appropriate C library file or files to resolve the references to the library functions and provide the code for the called library functions. The procedures for linking with the C library are discussed in detail in the *Microsoft C Compiler User's Guide*.

In most cases you must prepare for the call to the run-time library routine by performing one or both of these steps.

1. Including a given file in your program. Many routines require definitions and declarations that are provided by an include file.
2. Providing declarations for library functions that return values of any type but **int**. The compiler expects all functions to have **int** return type unless declared otherwise. You can provide these declarations by including the C library file containing the declarations or by explicitly declaring the functions within your program.

These are the minimum steps required; you may also want to take other steps, such as enabling type-checking for the arguments in function calls.

The remainder of this chapter discusses the preparation procedures for using run-time library routines and special rules (such as filename and pathname conventions) that may apply to some routines.

## 2.2 Identifying Functions and Macros

Most of the routines in the C run-time library are C functions; that is, they consist of compiled C statements. However, some routines are implemented as "macros". A macro is an identifier defined with the C preprocessor directive **#define** to represent a value or expression. Like a function, a macro can be defined to take zero or more arguments, which replace formal parameters in the macro definition. Defining and using macros is discussed in detail in the *Microsoft C Language Reference*.

The macros defined in the C run-time library behave like functions: they take arguments and return values, and they are invoked in a similar manner. The major advantage of using macros is faster execution time; their definitions are expanded in the preprocessing stage, eliminating the overhead required for a function call. However, because macros are expanded (replaced by their definitions) before compilation, they can increase the size of a program, particularly when there are multiple occurrences of the macro in the program. Unlike a function, which is defined only once regardless of how many times it is called, each occurrence of a macro is expanded. Functions and macros thus offer a trade-off between speed and size. In several cases, the C library provides both macro and function versions of the same library routine to allow you this choice.

Some important differences between functions and macros are described in the following list.

1. Some macros may treat arguments with side effects incorrectly when the macro is defined so that arguments are evaluated more than once. See the example that follows this list.
2. A macro identifier does not have the same properties as a function identifier. In particular, a macro identifier does not evaluate to an address, as a function identifier does. You cannot, therefore, use a macro identifier in contexts requiring a pointer. For instance, if you give a macro identifier as an argument in a function call, the *value* represented by the macro is passed; if you give a function identifier as an argument in a function call, the *address* of the function is passed.
3. Since macros are not functions, they cannot be declared, nor can pointers to macro identifiers be declared. Thus, type-checking cannot be performed on macro arguments. The compiler does, however, detect cases where the wrong number of arguments is specified for the macro.
4. The library routines implemented as macros are defined through preprocessor directives in the library include files. To use a library macro, you must include the appropriate file, or the macro will be undefined.

The routines that are implemented as macros are marked with a note in Part 2, "Reference," of this manual. You can examine a particular macro definition in the corresponding include file to determine whether arguments with side effects will cause problems.

### Example

```
#include <ctype.h>

int a = 'm';
a = toupper(a++);
```

This example uses the **toupper** routine from the standard C library. The **toupper** routine is implemented as a macro; its definition in *ctype.h* is

```
#define toupper(c)     ((islower(c)) ? _toupper(c) : (c))
```

The definition uses the conditional operator (? :). In the conditional expression, the argument *c* is evaluated twice: once to determine whether or not it is lowercase, and once to return the appropriate result. This causes the argument "a++" to be evaluated twice, thus increasing *a* twice rather than once. As a result, the value operated on by **islower** differs from the value operated on by **_toupper**.

Not all macros have this effect; you can determine whether a macro will handle side effects properly by examining the macro definition before using it.

## 2.3 Including Files

Many run-time routines use macros, constants, and types that are defined in separate include files. To use these routines, you must incorporate the specified file (using the preprocessor directive **#include**) into the source file being compiled.

The contents of each include file are different, depending on the needs of specific run-time routines. However, in general, include files contain combinations of the following.

- *Definitions of manifest constants.* For example, the constant **BUFSIZ**, which determines the size of buffers for buffered input and output operations, is defined in *stdio.h*.
- *Definitions of types.* Some run-time routines take data structures as arguments or return values with structure types. Include files set up the required structure type definitions. For example, most stream input and output operations use pointers to a structure of type **FILE**, defined in *stdio.h*.

- *Two sets of function declarations.* The first set of declarations gives return types and argument type lists for run-time functions, while the second set declares only the return type. Declaring the function return type is required for any function that returns a value with type other than int; see Section 2.4, "Declaring Functions." The presence of an argument type list enables type-checking for the arguments in a function call; see Section 2.5, "Argument Type-Checking," for a discussion of this option.
- *Macro definitions.* Some routines in the run-time library are implemented as macros. The definitions for these macros are contained in the include files. To use one of these macros, you must include the appropriate file.

The reference page for each library routine lists the include file or files needed by the routine.

## 2.4 Declaring Functions

Whenever you use a library function that returns any type of value but an int, you should make sure that the function is declared before it is called. The easiest way to do this is to include the file containing declarations for that function, causing the appropriate declarations to be placed in your program.

Two sets of function declarations are provided in each include file. The first set declares both the function return type and the argument type list for the function. This set is included only when you enable argument type-checking, as described in Section 2.5. Use of the argument type-checking feature is highly recommended, since mismatches between actual and formal arguments to a function can cause serious and possibly hard-to-detect errors.

The second set of function declarations declares only the function return type. This set is included when argument type-checking is *not* enabled.

Your program can contain more than one declaration of the same function, as long as the declarations are compatible. This is an important feature to remember if you have older programs whose function declarations do not contain argument type lists. For instance, if your program contains the declaration

```
char *calloc();
```

you can also include the declaration

```
char *calloc(unsigned, unsigned);
```

Although the two declarations are not identical, they are compatible, so no conflict occurs.

You may provide your own function declarations instead of using the declarations in the library include files if you wish. It is recommended, however, that you consult the declarations in the include files to make sure that your declarations are correct.

## 2.5 Argument Type-Checking

The Microsoft C Compiler offers a type-checking feature for the arguments in a function call. Type-checking is performed whenever an argument type list is present in a function declaration. The form of the argument type list and the type-checking method are discussed in full in the *Microsoft C Language Reference.*

For functions that you write yourself, you are responsible for setting up argument type lists to invoke type-checking. You can also use the /Zg command line option to cause the compiler to generate a list of function declarations for all functions defined in a particular source file; the list may then be incorporated into your program. See the *Microsoft C Language Reference* for details.

For functions in the C run-time library, you can use the procedures outlined in this section to perform type-checking on arguments. Every function in the C run-time library is declared in one of the library include files. Two declarations are given for each function: one with and one without an argument type list. The function declarations are enclosed in an **#ifdef** preprocessor block. If you define an identifier named **LINT_ARGS**, the declarations containing argument type lists are processed and compiled, thus enabling argument type-checking. If the **LINT_ARGS** identifier is not defined, the declarations without argument type lists are included, and argument type-checking will not be performed.

By default, **LINT_ARGS** is undefined, so no type-checking is performed for library functions. You can define **LINT_ARGS** in one of two ways:

1. Use the /D command-line option to define **LINT_ARGS** at compile time.
2. Define **LINT_ARGS** with a **#define** directive in your source file. The **#define** directive must occur *before* the **#include** directive for the given file to be effective.

The value of **LINT_ARGS** is not significant; you can define it to any value, including an empty value.

Notice that the **LINT_ARGS** definition applies only to the library function declarations given in the include files. The function declarations in your source program or in your own include files are not affected. You can make the inclusion of your own declarations dependent on the **LINT_ARGS** identifier by using an **#if** or **#ifdef** directive. Refer to the library include files for a model.

Only limited type-checking can be performed on functions that take a variable number of arguments. The following run-time functions are affected by this limitation.

- In calls to **cprintf**, **cscanf**, **printf**, and **scanf**, type-checking is performed only on the first argument: the format string.
- In calls to **fprintf**, **fscanf**, **sprintf**, and **sscanf**, type-checking is performed on the first two arguments: the file or buffer and the format string.
- In calls to **open**, only the first two arguments are type-checked: the pathname and open flags.
- In calls to **sopen**, the first three arguments are type-checked: the pathname, open flags, and sharing mode.
- In calls to **execl**, **execle**, and **execlp**, type-checking is performed on the first two arguments: the pathname and the first argument pointer.
- In calls to **spawnl**, **spawnle**, and **spawnp**, type-checking is performed on the first three arguments: the mode flag, the pathname, and the first argument pointer.

## 2.6 Error Handling

When calling a function, it is a good idea to provide for detection and handling of error returns, if any. Otherwise, your program may produce unexpected results.

For run-time library functions, you can determine the expected return value from the return value discussion on each library page. In some cases no established error return exists for a routine. This usually occurs when the range of legal return values makes it impossible to return a unique error value.

The discussion of some routines indicates that when an error occurs, a global variable named **errno** is set to a value indicating the type of error. Notice that you cannot depend upon **errno** being set unless the description of the routine explicitly mentions the **errno** variable.

When using routines that set **errno**, you can test the **errno** values against the error values defined in *errno.h*, or you can use the **perror** routine to print the system error message to standard error (**stderr**). For a listing of **errno** values and the associated error messages, see Appendix A, "Error Messages."

When you use **errno** and **perror**, keep in mind that the value of **errno** reflects the error value for the last call that set **errno**. To prevent misleading results, you should always test the return value to verify that an error actually occurred before you access **errno**. Once you determine that an error occurred, you should use **errno** or **perror** immediately. Otherwise, the value of **errno** may be changed by intervening calls.

The math routines set **errno** upon error in the manner described on the reference page for each math function in Part 2 of this manual. Math routines handle errors by invoking a function named **matherr**. You can choose to handle math errors differently by writing your own error routine and naming it **matherr**. When you provide your own **matherr** function, that function is used in place of the run-time library version. You must follow certain rules when writing your own **matherr** function, as outlined on the **matherr** reference page in Part 2 of this manual.

You can check for errors in stream operations by calling the **ferror** routine. The **ferror** routine detects whether the error indicator for a given stream has been set. The error indicator is cleared automatically when the stream is closed or rewound, or the **clearerr** function can be called to reset the error indicator.

Errors in low-level input and output operations cause **errno** to be set.

The **feof** routine tests for end-of-file on a given stream. An end-of-file condition in low-level input and output can be detected with the **eof** routine or when a **read** operation returns zero as the number of bytes read.

## 2.7 Filenames and Pathnames

Many routines in the run-time library accept strings representing pathnames and filenames as arguments. The routines process the arguments and pass them to the operating system, which is ultimately responsible for creating and maintaining files and directories. Thus, it is important to keep in mind not only the C conventions for strings, but also the operating system rules for filenames and pathnames and the differences between MS-DOS and XENIX rules. There are several considerations:

1. Case-sensitivity
2. Subdirectory conventions
3. Delimiters for pathname components

The C language is case-sensitive, meaning that it distinguishes between uppercase and lowercase letters. The MS-DOS operating system is not case-sensitive. When accessing files and directories on MS-DOS, you cannot use case differences to distinguish between identical names. For example, the names "FILEA" and "fileA" are equivalent and refer to the same file.

Portability considerations may also affect how you choose filenames and pathnames. For instance, if you plan to port your code to a XENIX system, you should take the XENIX naming conventions into account. Unlike MS-DOS, XENIX is case-sensitive. Thus, the following two directives are equivalent on MS-DOS but not on XENIX.

```
#include <STDIO.H>
#include <stdio.h>
```

To produce portable code, you should use the name that works correctly on XENIX, since either case works on MS-DOS.

The convention of storing some include files in a subdirectory named "sys" is also a XENIX convention. The convention is adopted in this manual, which includes the "sys" subdirectory in the specification for the appropriate include files. If you're not concerned with portability, you can disregard this convention and set up your include files accordingly. If you are concerned with portability, using the "sys" subdirectory can make portability between XENIX and MS-DOS easier.

The MS-DOS and XENIX operating systems differ in the handling of pathname delimiters. XENIX uses the forward slash (/) to delimit the components of pathnames, while MS-DOS ordinarily uses the backslash (\). However, MS-DOS is able to recognize the forward slash (/) as a delimiter internally in situations where a pathname is expected. Thus, you can use either a backslash or a forward slash in MS-DOS pathnames within C programs, as long as the context is unambiguous and a pathname is clearly expected.

This rule applies to most of the routines in the run-time library. Wherever a pathname argument is required, you can use either a forward slash or a backslash as a delimiter. If you are concerned with portability to XENIX, you should use the forward slash.

However, the exceptions to the rule are important. The following routines accept string arguments that are not known in advance to be pathnames (they may be pathnames, but are not required to be). In these cases, the arguments are treated as C strings, and special rules apply.

- In the **exec** and **spawn** families of routines, you pass the name of a program to be executed as a child process and then pass strings representing arguments to the child process. The pathname of the program to be executed as the child process can use either forward slashes or backslashes as delimiters, since a pathname argument is expected. However, it is recommended that you use backslashes in any pathname arguments to the child process, since the program being executed as the child process may simply expect a string argument that is not necessarily a pathname.
- In the **system** call you pass a command to be executed by MS-DOS; this command may or may not include a pathname.

In these cases, only the backslash (\) separator should be used as a pathname delimiter. The forward slash (/) will not be recognized. However, in C strings, the backslash is an escape character. It signals that a special escape sequence follows. If an ordinary character follows the backslash, the backslash is disregarded and the character is printed. Thus, the

sequence "\\" is required to produce a single backslash in a C string. (See your *Microsoft C Language Reference* for a full discussion of escape sequences.)

When you want to pass a pathname argument to the child process in an **exec** or **spawn** call, or when you use a pathname in a **system** call, you must use the double backslash sequence (\\) to represent a single pathname delimiter.

## Examples

```
1.  result = system("DIR B:\\TOP\\DOWN");

2.  spawnl(P_WAIT, "bin/show", "4", "sub", "bin\\tell", NULL);
```

In the first example, double backslashes must be used in the call to **system** to represent the pathname "DIR B:\TOP\DOWN". Note that not all calls to **system** use a pathname; for example,

```
result = system("DIR");
```

does not contain a pathname.

In the second example, the **spawnl** routine is used to execute the file named SHOW.EXE in the BIN subdirectory. Since a pathname is expected as the second argument, the forward slash can be used. (A single backslash would also be acceptable.) The first two arguments passed to SHOW.EXE are the strings "4" and "sub". The third argument is a string representing a pathname. The sequence "\\" must be used to represent a single backslash between "bin" and "tell".

# 2.8 Binary and Text Modes

Most C programs use one or more data files for input and output. Under MS-DOS, data files are ordinarily processed in "text" mode. In text mode, carriage return-linefeed combinations (CR-LF) are translated into a single linefeed (LF) character on input. Linefeed characters are translated to carriage return-linefeed combinations on output.

In some cases you may want to process files without making these translations. In binary mode, carriage return-linefeed translations are suppressed.

You can control the translation mode for the files used in a program in the following ways.

- To process a few selected files in binary mode, while retaining the default text mode for most files, you can specify binary mode when you open the selected files. The **fopen** routine opens a file in binary mode when the letter "b" is specified in the access *type* string for the file. If you use the **open** routine, you can specify the **O_BINARY** flag in the *oflag* argument to cause the file to be opened in binary mode. See the reference pages for these routines in Part 2 of this manual for details.

- To process most or all files in binary mode, you can change the default mode to binary. The global variable **_fmode** controls the default translation mode. When **_fmode** is set to **O_BINARY**, the default mode is binary; otherwise, the default mode is text, except for **stdaux** and **stdprn**, which are opened in binary mode by default. The initial setting of **_fmode** is text, by default.

  You can change the value of **_fmode** in one of two ways. First, you can link with the file BINMODE.OBJ (supplied with your compiler software). Linking with BINMODE.OBJ changes the initial setting of **_fmode** to **O_BINARY**, causing all files except **stdin**, **stdout**, and **stderr** to be opened in binary mode. This option is described in the *Microsoft C Compiler User's Guide.*

  Second, you can change the value of **_fmode** directly, by setting it to **O_BINARY** in your program. This has the same effect as linking with BINMODE.OBJ.

  You can still override the default mode (now binary) for particular files by opening them in text mode. The **fopen** routine opens a file in text mode when the letter "t" is specified in the access *type* string for the file. If you use the **open** routine, you can specify the **O_TEXT** flag in the *oflag* argument to cause the file to be opened in text mode. See the reference pages for these routines for details.

- The **stdin**, **stdout**, and **stderr** streams are opened in text mode by default; **stdaux** and **stdprn** are opened in binary mode. To process **stdin**, **stdout**, or **stderr** in binary mode instead, or to process **stdaux** or **stdprn** in text mode, use the **setmode** routine. This routine can also be used to change the mode of a file after it has been opened. The **setmode** routine takes two arguments, a file handle and a

translation mode argument, and sets the mode of the file accordingly.

## 2.9 MS-DOS Considerations

The use of some routines in the run-time library is affected by the version of MS-DOS you are using. These routines are listed and described below.

**dosexterr, locking, sopen**

These three routines are effective only on MS-DOS Versions 3.0 and later. The **sopen** function opens a file with file-sharing attributes; this function should be used in place of **open** when you want a file to have such attributes. The **locking** function locks all or part of a file from access by other users. The **dosexterr** function provides error-handling for system call 59H in MS-DOS Versions 3.0 and later.

**dup, dup2**

In certain cases, using the **dup** and **dup2** functions on versions of MS-DOS earlier than 3.0 may cause unexpected results. When you use **dup** or **dup2** to create a duplicate file handle for **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** under versions of MS-DOS earlier than 3.0, calling the **close** function with either handle causes errors in later I/O operations using the other handle. Under MS-DOS Versions 3.0 and later, the **close** is handled correctly and does not cause later errors.

**exec, spawn**

When using the **exec** and **spawn** families of routines under versions of MS-DOS earlier than 3.0, the value of the *arg0* or *arg*[0] argument is not available to the user; a null string is stored in that position. Under MS-DOS Version 3.0 and later, the value of *arg0* or *arg*[0] is available to the user.

To write programs that will run on all versions of MS-DOS, you can use the **_osmajor** and **_osminor** variables (discussed in Section 3.5 of Chapter 3, "Global Variables and Standard Types") to test the current operating system version number and take the appropriate action based on the result of the test.

### Example

```
unsigned char _osmajor;
   .
   .
   .
if (_osmajor > 2)
        open("TEST.DAT", O_RDWR);
else
        sopen("TEST.DAT", O_RDWR, SH_DENYWR);
```

In this example, the global variable **_osmajor** is tested to determine whether the file TEST.DAT should be opened using the **open** routine (under versions of MS-DOS earlier than 3.0) or the **sopen** routine (MS-DOS Versions 3.0 and later).

## 2.10 Floating-Point Support

The math routines supplied in the C run-time library require floating-point support to perform calculations with real numbers. This support can be provided by the floating-point libraries that accompany your compiler software or by an 8087 or 80287 coprocessor. (For information on selecting and using a floating-point library with your program, see the *Microsoft C Compiler User's Guide*.) The names of the routines that require floating-point support are listed below.

| | | | | |
|---|---|---|---|---|
| **acos** | **cabs** | **fabs** | **hypot** | **sin** |
| **asin** | **ceil** | **fcvt** | **idexp** | **sinh** |
| **atan** | **cos** | **floor** | **log** | **sqrt** |
| **atan2** | **cosh** | **fmod** | **log10** | **tan** |
| **atof** | **ecvt** | **frexp** | **modf** | **tanh** |
| **bessel†** | **exp** | **gcvt** | **pow** | |

† Note: **bessel** does not correspond to a single function but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

In addition, the **printf** family of routines (**cprintf, fprintf, printf,** and **sprintf**) require support for floating-point input and output if they are used to print floating-point values.

The C compiler tries to detect whether floating-point values are used in a program so that supporting routines are loaded only if required. This behavior allows a considerable space savings for programs that do not require floating-point support.

When you use a floating-point type character in the format string for the **printf** or **scanf** functions (**cprintf, fprintf, printf, sprintf, cscanf, fscanf, scanf,** or **sscanf**), make sure that you specify floating-point values or pointers to floating-point values in the argument list to correspond to any floating-point type characters in the format string. The presence of floating-point arguments allows the compiler to detect the use of floating-point values. If a floating-point type character is used to print, for example, an integer argument, the use of floating-point values will not be detected because the compiler does not actually read the format string used in the **printf** and **scanf** functions. For instance, the following program causes a run-time error.

```
main()                  /* THIS EXAMPLE PRODUCES AN ERROR */
{
        long f = 10L;
        printf("%f", f);
}
```

This program produces the following message at run time.

```
Floating point not loaded
```

The use of floating-point values is not detected because no floating-point arguments are given in the call to **printf**. Consequently, the routines required for floating-point input and output are not loaded, and an error occurs.

The following is a corrected version of the above call to **printf**.

```
printf("%f", (double)f);                /* CORRECTED VERSION */
```

This version corrects the error by casting the long integer value to **double** type.

# Chapter 3
# Global Variables and Standard Types


3.1 Introduction 29
3.2 daylight, timezone, tzname 29
3.3 _doserrno, errno, sys_errlist, sys_nerr 30
3.4 _fmode 31
3.5 _osmajor, _osminor 31
3.6 environ, _psp 32
3.7 Standard Types 33

# 3.1 Introduction

The C run-time library contains definitions for a number of variables and types used by library routines. You can access these variables and types by including the files in which they are declared or by giving appropriate declarations in your program, as shown in the following sections.

# 3.2 daylight, timezone, tzname

## Declarations

```
int daylight;
long timezone;
char *tzname[2];
```

The **daylight**, **timezone**, and **tzname** variables are used by several of the time and date functions to make local time adjustments and are declared in the include file *time.h*. The values of the variables are determined by the setting of an environment variable named TZ.

You can control local time adjustments by setting the TZ environment variable. The value of the environment variable TZ must be a three-letter time zone, followed by a possibly signed number giving the difference in hours between Greenwich Mean Time and local time. The number is positive moving west from Greenwich, negative moving east. The number may be followed by a three-letter Daylight Savings Time zone. For example, the command

```
SET TZ=EST5EDT
```

specifies that the local time zone is EST (Eastern Standard Time), that local time is 5 hours earlier than Greenwich Mean Time, and that Daylight Savings Time (EDT) is in effect. Omitting the Daylight Savings Time zone, as shown below, means that no corrections will be made for Daylight Savings Time.

```
SET TZ=EST5
```

When you call the **ftime** or **localtime** function, the values of the three variables **daylight**, **timezone**, and **tzname** are determined from the TZ setting. The **daylight** variable is given a nonzero value if a Daylight Savings Time zone is present in the TZ setting; otherwise, **daylight** is zero. The **timezone** variable is assigned the difference in seconds (calculated by converting the hours given in the TZ setting) between Greenwich Mean Time and local time. The first element of the **tzname** variable is the string value of the three-letter time zone from the TZ setting; the second element is the string value of the Daylight Savings Time zone. If the Daylight Savings Time zone is omitted from the TZ setting, **tzname**[1] is an empty string.

If you do not explicitly assign a value to TZ before calling **ftime** or **localtime**, the following default setting is used.

```
PST8PDT
```

The **ftime** and **localtime** functions call another function, **tzset**, to assign values to the three global variables from the TZ setting. You can also call **tzset** directly if you like; see the **tzset** reference page in Part 2 of this manual for details.

## 3.3 _doserrno, errno, sys_errlist, sys_nerr

```
int _doserrno;
int errno;
char *sys_errlist[];
int sys_nerr;
```

The **errno**, **sys_errlist**, and **sys_nerr** variables are used by the **perror** function to print error information and are declared in the include file *stdlib.h*. When an error occurs in a system-level call, the **errno** variable is set to an integer value to reflect the type of error. The **perror** function uses the **errno** value to look up (index) the corresponding error message in the **sys_errlist** table. The value of the **sys_nerr** variable is defined as the number of elements in the **sys_errlist** array. For a listing of the **errno** values and the corresponding error messages, see Appendix A, "Error Messages."

The **errno** values on MS-DOS are a subset of the values for **errno** on XENIX systems. Thus, the value assigned to **errno** in case of error does not necessarily correspond to the actual error code returned by an MS-DOS system call. Instead, the actual MS-DOS error codes are mapped onto the **perror** values. If you want to access the actual MS-DOS error code, you can use the **_doserrno** variable. When an error occurs in a system call, the **_doserrno** variable is assigned the actual error code returned by the corresponding MS-DOS system call. (See the *Microsoft MS-DOS Programmer's Reference Manual* for details on MS-DOS error returns.)

In general, you should only use **_doserrno** for error detection in operations involving input and output, since the **errno** values for input and output errors have MS-DOS error code equivalents. Not all of the error values available for **errno** have exact MS-DOS error code equivalents, and some may have no equivalents, causing the value of **_doserrno** to be undefined.

## 3.4 _fmode

```
int _fmode;
```

The **_fmode** variable controls the default file translation mode. It is declared in *stdlib.h*. By default, the value of **_fmode** is zero, causing files to be translated in text mode (unless specifically opened or set to binary mode). When **_fmode** is set to **O_BINARY**, the default mode is binary. You can set **_fmode** to **O_BINARY** by linking with BINMODE.OBJ or by assigning it the value **O_BINARY**. See Section 2.8, "Binary and Text Modes," in Chapter 2, "Using the C Library Routines," for a discussion of file translation modes and the use of the **_fmode** variable.

## 3.5 _osmajor, _osminor

```
unsigned char _osmajor;
unsigned char _osminor;
```

The **_osmajor** and **_osminor** variables provide information about the version number of MS-DOS currently in use. They are declared in *stdlib.h*. The **_osmajor** variable holds the "major" version number. For example, under MS-DOS Version 2.0, **_osmajor** is equal to 2, while under MS-DOS Version 3.0, **_osmajor** is 3.

The **_osminor** variable stores the "minor" version number. For example, under MS-DOS Version 2.0, **_osminor** is 0 (zero), while under MS-DOS Version 2.1, **_osminor** is 1.

These variables can be useful when you want to write code to run on different versions of MS-DOS. For example, you can test the **_osmajor** variable before making a call to **sopen**; if the major version number is earlier (less) than 3, **open** should be used instead of **sopen**.

## 3.6 environ, _psp

```
char *environ[];
unsigned int _psp;
```

The **environ** and **_psp** variables provide access to memory areas containing process-specific information. Both variables are declared in the include file *stdlib.h*.

The **environ** variable is a pointer to the array of strings making up the process environment. The environment consists of one or more entries of the form

*name=string*

where *name* is the name of an environment variable and *string* is the value of that variable. The *string* may be empty. The initial environment settings are taken from the MS-DOS environment at the time of the program's execution.

The **getenv** and **putenv** routines use the **environ** variable to access and modify the environment table. When **putenv** is called to add or delete environment settings, the environment table changes in size, and its location in memory may also change, depending on the program's memory requirements. The **environ** variable is adjusted in these cases and will always point to the correct table location.

The **_psp** variable contains the segment value of the Program Segment Prefix (PSP) for the process. The PSP contains execution information about the process, such as a copy of the command line that invoked the process and the return address for process terminate or interrupt. (See your *Microsoft MS-DOS Programmer's Reference Manual* for details.) The **_psp** variable can be used to form a long pointer to the PSP (**_psp**:0).

## 3.7 Standard Types

A number of run-time library routines use structure values whose types are defined in include files. These types are listed and described as follows, and the include file that defines each type is given. For a listing of the actual structure definitions, see the description of the appropriate include file in Chapter 5, "Include Files."

| Standard Type | Description |
|---|---|
| **complex** | The **complex** structure, defined in *math.h*, stores the real and imaginary parts of a complex number and is used by the **cabs** function. |
| **DOSERROR** | The **DOSERROR** structure, defined in *dos.h*, stores values returned by the MS-DOS system call 59H (available under MS-DOS Versions 3.0 and later). |
| **exception** | The **exception** structure, defined in *math.h*, stores error information for math routines and is used by the **matherr** routine. |
| **FILE** | The **FILE** structure, defined in *stdio.h*, is the structure used in all stream input and output operations. The fields of the **FILE** structure store information about the current state of the stream. |
| **jmp_buf** | The **jmp_buf** type, declared in *setjmp.h*, is an array type rather than a structure type. It defines the buffer used by the **setjmp** and **longjmp** routines to save and restore the program environment. |
| **REGS** | The **REGS** union, defined in *dos.h*, stores byte and word register values to be passed to and returned from calls to the MS-DOS interface functions. |
| **SREGS** | The **SREGS** structure, defined in *dos.h*, stores the values of the ES, CS, SS, and DS registers. This structure is used by the MS-DOS interface functions that require segment register values (**int86x**, **intdosx**, and **segread**). |

| | |
|---|---|
| **stat** | The **stat** structure, defined in *sys\stat.h*, contains file status information returned by the **stat** and **fstat** routines. |
| **timeb** | The **timeb** structure, defined in *sys\timeb.h*, is used by the **ftime** routine to store the current system time in a broken-down format. |
| **tm** | The **tm** structure, defined in *time.h*, is used by the **asctime**, **gmtime**, and **localtime** functions to store and retrieve time information. |
| **utimbuf** | The **utimbuf** structure, defined in *sys\utime.h*, stores file access and modification times used by the **utime** function to change file modification dates. |

# Chapter 4
# Run-Time Routines by Category


4.1 Introduction 37
4.2 Buffer Manipulation 37
4.3 Character Classification and Conversion 38
4.4 Data Conversion 40
4.5 Directory Control 40
4.6 File Handling 41
4.7 Input and Output 42
4.7.1 Stream Routines 43
4.7.1.1 Opening a Stream 45
4.7.1.2 Predefined Stream Pointers: stdin, stdout, stderr, stdaux, stdprn 46
4.7.1.3 Controlling Stream Buffering 47
4.7.1.4 Closing Streams 48
4.7.1.5 Reading and Writing Data 48
4.7.1.6 Detecting Errors 49
4.7.2 Low-Level Routines 49
4.7.2.1 Opening a File 50
4.7.2.2 Predefined Handles 50
4.7.2.3 Reading and Writing Data 51
4.7.2.4 Closing files 51
4.7.3 Console and Port I/O Routines 52

4.8 Math 54
4.9 Memory Allocation 55
4.10 MS-DOS Interface 57
4.11 Process Control 59
4.12 Searching and Sorting 61
4.13 String Manipulation 62
4.14 Time 63
4.15 Miscellaneous 64

# 4.1 Introduction

This chapter describes the major categories of routines included in the C run-time libraries. The discussions of these categories are intended to give a brief overview of the capabilities of the run-time library. For a complete description of the syntax and use of each routine, see Part 2, "Reference," of this manual.

# 4.2 Buffer Manipulation

| Routine | Use |
|---|---|
| memccpy | Copies characters from one buffer to another, until a given character is copied or a given number of characters has been copied |
| memchr | Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer |
| memcmp | Compares a specified number of characters from two buffers |
| memcpy | Copies a specified number of characters from one buffer to another |
| memset | Uses a given character to initialize a specified number of bytes in the buffer |
| movedata | Copies a specified number of characters from one buffer to another, even when buffers are in different segments |

The buffer manipulation routines are useful for working with areas of memory on a character-by-character basis. Buffers are arrays of characters (bytes). However, unlike strings, they are not usually terminated with a null character ('\0'). Thus, the buffer manipulation routines always take a length or count argument.

Function declarations for the buffer manipulation routines are given in the include file *memory.h*.

# 4.3 Character Classification and Conversion

| Routine | Use |
|---|---|
| isalnum | Tests for alphanumeric character |
| isalpha | Tests for alphabetic character |
| isascii | Tests for ASCII character |
| iscntrl | Tests for control character |
| isdigit | Tests for decimal digit |
| isgraph | Tests for printable character except space |
| islower | Tests for lowercase character |
| isprint | Tests for printable character |
| ispunct | Tests for punctuation character |
| isspace | Tests for whitespace character |
| isupper | Tests for uppercase character |
| isxdigit | Tests for hexadecimal digit |
| toascii | Converts character to ASCII code |
| tolower | Tests character and converts to lowercase if uppercase |
| toupper | Tests character and converts to uppercase if lowercase |
| _tolower | Converts character to lowercase (unconditional) |
| _toupper | Converts character to uppercase (unconditional) |

The character classification and conversion routines let you test individual characters in a variety of ways, and convert between uppercase and lowercase characters. The classification routines identify a character by looking it up in a table of classification codes; using these routines is generally faster than writing an equivalent test expression (such as "if ((c >= 0) | | c <= 0x7f)") to classify a character.

The **tolower** and **toupper** routines are implemented both as functions and as macros; the remainder of the routines in this category are implemented only as macros. All of the macros are defined in *ctype.h*, and this file must be included or the macros will be undefined.

The **toupper** and **tolower** macros evaluate their argument twice and thus cause arguments with side effects to give incorrect results. For this reason, you may want to use the function versions of these routines instead.

The macro versions of **tolower** and **toupper** are used by default when you include *ctype.h*. To use the function versions instead, you must give **#undef** preprocessor directives for **tolower** and **toupper** *after* the **#include** directive for *ctype.h* but *before* you call the routines. This procedure removes the macro definitions and causes occurrences of **tolower** and **toupper** to be treated as function calls to the **tolower** and **toupper** library functions.

If you want to use the function versions of **toupper** and **tolower** and you do not use any of the other character classification macros in your program, you can simply omit the *ctype.h* include file. In this case no macro definitions are present for **tolower** and **toupper**, so the function versions will be used.

Function declarations for the **tolower** and **toupper** functions are given in the include file *stdlib.h* instead of *ctype.h* to avoid conflict with the macro definitions. When you want to use **tolower** and **toupper** as functions and include the declarations from *stdlib.h*, you must follow this sequence.

1. Include *ctype.h* if required for other macro definitions.
2. If *ctype.h* was included, give **#undef** directives for **tolower** and **toupper.**
3. Include *stdlib.h*.

The declarations of **tolower** and **toupper** in *stdlib.h* are enclosed in an **#ifndef** block and are processed only if the corresponding identifier (**toupper** or **tolower**) is not defined.

## 4.4 Data Conversion

The data conversion routines convert

| Routine | Use |
|---|---|
| **atof** | Converts string to **float** |
| **atoi** | Converts string to **int** |
| **atol** | Converts string to **long** |
| **ecvt** | Converts **double** to string |
| **fcvt** | Converts **double** to string |
| **gcvt** | Converts **double** to string |
| **itoa** | Converts **int** to string |
| **ltoa** | Converts **long** to string |
| **ultoa** | Converts **unsigned long** to string |

The data conversion routines convert numbers to strings of ASCII characters and vice versa. These routines are implemented as functions, and all except **atof** are declared in the include file *stdlib.h*. The **atof** function, which converts a string to a floating-point value, is declared in *math.h*.

## 4.5 Directory Control

| Routine | Use |
|---|---|
| **chdir** | Changes current working directory |
| **getcwd** | Gets current working directory |
| **mkdir** | Makes a new directory |
| **rmdir** | Removes a directory |

The directory control routines let you access, modify, and obtain information about the directory structure from within your program. You can get the current working directory, change directories, and add or remove directories.

The directory routines are functions and are declared in the include file *direct.h*.

## 4.6 File Handling

| Routine | Use |
|---|---|
| **access** | Checks file permission setting |
| **chmod** | Changes file permission setting |
| **chsize** | Changes file size |
| **filelength** | Checks file length |
| **fstat** | Gets file status information on handle |
| **isatty** | Checks for character device |
| **locking** | Locks areas of file (available with MS-DOS Version 3.0 and later) |
| **mktemp** | Creates a unique filename |
| **rename** | Renames file |
| **setmode** | Sets file translation mode |
| **stat** | Gets file status information on named file |
| **umask** | Sets default permission mask |
| **unlink** | Deletes a file |

The file handling routines work on a file designated by a pathname or file handle. They modify or give information about the designated file. All of these routines except **fstat** and **stat** are declared in the include file *io.h*. The **fstat** and **stat** functions are declared in *sys\stat.h*.

The **access**, **chmod**, **rename**, **stat**, and **unlink** routines operate on files specified by a pathname or filename.

The **chsize**, **filelength**, **isatty**, **locking**, **setmode**, and **fstat** routines work with files designated by a file handle. The **locking** routine works only under MS-DOS Version 3.0 and later; it locks a region of a file against access by other users.

The **mktemp** and **umask** routines have slightly different functions than the above routines. The **mktemp** routine creates a unique filename. Programs can use **mktemp** to create unique filenames that do not conflict with the names of existing files. The **umask** routine sets the default permission mask for any new files created in a program. The mask may override the permission setting given in the **open** or **creat** call for the new file.

## 4.7 Input and Output

The input and output routines of the standard C library allow you to read and write data to and from files and devices. In C, there are no predefined file structures; all data are treated as sequences of bytes. Three types of input and output (I/O) functions are available:

1. Stream I/O
2. Low-level I/O
3. Console and port I/O

The "stream" functions treat a data file or data item as a stream of individual characters. By choosing among the many stream functions available, you can process data in different sizes and formats, from single characters to large data structures.

When a file is opened for I/O using the stream functions, the opened file is associated with a structure of type **FILE** (defined in *stdio.h*) containing basic information about the file. A pointer to the **FILE** structure is returned when the stream is opened. This pointer (also called the stream pointer, or just *stream*) is used to refer to the file in subsequent operations.

The stream functions provide for (optionally) buffered and formatted input and output. When a stream is buffered, data that is read from or written to the stream is collected in an intermediate storage location called a buffer. When writing, the output buffer's contents are written to the appropriate final location when the buffer is full. The buffer is said to be "flushed" when this occurs. When reading, a block of data is placed in the input buffer and data are read from the buffer; when the input buffer is empty, the next block of data is transferred into the buffer.

Buffering produces efficient I/O because the system can transfer a large block of data in a single operation rather than performing an I/O operation each time a data item is read from or written to a stream. However, if a program terminates abnormally, output buffers may not be flushed, resulting in loss of data.

The console and port I/O routines can be considered an extension of the stream routines. They allow you to read or write to a console (terminal) or an input/output port (such as a printer port). The port I/O routines simply read and write data in bytes. Some additional options are available with console I/O routines. For example, you can detect whether a character has been typed at the console. You can also choose between echoing characters to the screen as they are read, or reading characters without echoing.

The "low-level" input and output routines do not perform buffering and formatting; they may be considered as invoking the operating system's input and output capabilities directly. These routines let you access files and peripheral devices at a more basic level than the stream functions.

When a file is opened with a low-level routine, a file *handle* is associated with the opened file. This handle is an integer value that is used to refer to the file in subsequent operations.

---

*Warning*

Stream routines and low-level routines are generally incompatible, so either stream or low-level functions should be used consistently on a given file. Since stream functions are buffered and low-level functions are not, attempting to access the same file or device by two different methods causes confusion and may result in the loss of data in buffers.

---

## 4.7.1 Stream Routines

| Routine | Use |
|---|---|
| clearerr | Clears the error indicator for a stream |
| fclose | Closes a stream |

| | |
|---|---|
| **fcloseall** | Closes all open streams |
| **fdopen** | Opens a stream using a *handle* |
| **feof** | Tests for end-of-file on a stream |
| **ferror** | Tests for error on a stream |
| **fflush** | Flushes a stream |
| **fgetc** | Reads a character from *stream* (function version) |
| **fgetchar** | Reads a character from **stdin** (function version) |
| **fgets** | Reads a string from *stream* |
| **fileno** | Gets file handle associated with *stream* |
| **flushall** | Flushes all streams |
| **fopen** | Opens a stream |
| **fprintf** | Writes formatted data to *stream* |
| **fputc** | Writes a character to *stream* (function version) |
| **fputchar** | Writes a character to **stdout** (function version) |
| **fputs** | Writes a string to *stream* |
| **fread** | Reads fixed-length data items from *stream* |
| **freopen** | Reassigns a **FILE** pointer |
| **fscanf** | Reads formatted data from *stream* |
| **fseek** | Repositions file pointer to given location |
| **ftell** | Gets current file pointer position |
| **fwrite** | Writes fixed-length data items to **stdout** |
| **getc** | Reads a character from *stream* (macro version) |
| **getchar** | Reads a character from **stdin** (macro version) |
| **gets** | Reads a line from **stdin** |
| **getw** | Reads a binary **int** from *stream* |
| **printf** | Writes formatted data to **stdout** |
| **putc** | Writes a character to *stream* (macro version) |
| **putchar** | Writes a character to **stdout** (macro version) |

| | |
|---|---|
| **puts** | Writes a line to *stream* |
| **putw** | Writes a binary **int** from *stream* |
| **rewind** | Repositions file pointer to beginning of stream |
| **scanf** | Reads formatted data from **stdin** |
| **setbuf** | Controls stream buffering |
| **sprintf** | Writes formatted data to string |
| **sscanf** | Reads formatted data from string |
| **ungetc** | Places a character in the buffer |

To use the stream functions you must include the file *stdio.h* in your program. This file defines constants, types, and structures used in the stream functions, and contains function declarations and macro definitions for the stream routines.

Some of the constants defined in *stdio.h* may be useful in your program. The manifest constant **EOF** is defined to be the value returned at end-of-file. **NULL** is the null pointer. **FILE** is the structure that maintains information about a stream. **BUFSIZ** defines the size of stream buffers, in bytes.

### 4.7.1.1 Opening a Stream

A stream must be opened using the **fdopen, fopen**, or **freopen** function before input and output can be performed on that stream. When opening a stream, the named stream can be opened for reading, writing, or both and can be opened either in text or in binary mode.

The **fdopen, fopen**, and **freopen** functions return a **FILE** pointer, which is used to refer to the stream. When you call one of these functions, assign the return value to a **FILE** pointer variable and use that variable to refer to the opened stream. For example, if your program contains the line

```
infile = fopen ("test.dat", "r");
```

you can use the **FILE** pointer variable *infile* to refer to the stream.

### 4.7.1.2 Predefined Stream Pointers: stdin, stdout, stderr, stdaux, stdprn

When a program begins execution, five streams are automatically opened. These streams are the standard input, standard output, standard error, standard auxiliary, and standard print. By default, the standard input, standard output, and standard error refer to the user's console. This means that whenever a program expects input from the "standard input," it receives that input from the console. Similarly, a program that writes to the "standard output" prints its data to the console. Error messages generated by the library routines are sent to the "standard error," meaning that error messages appear on the user's console.

The assignment of the "standard auxiliary" and "standard print" streams depends on the machine configuration; these streams usually refer to an auxiliary port and a printer, respectively, but they may not be set up on a particular system. Be sure to check your machine configuration before using these streams.

When you use the stream functions, you can refer to the standard input, standard output, standard error, standard auxiliary, and standard print by using the following predefined **FILE** pointers.

| Stream | Device |
|---|---|
| **stdin** | Standard input |
| **stdout** | Standard output |
| **stderr** | Standard error |
| **stdaux** | Standard auxiliary |
| **stdprn** | Standard print |

You can use these pointers in any function that requires a stream pointer as an argument. Some functions, such as **getchar** and **putchar**, are designed to use **stdin** or **stdout** automatically. The pointers **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** are constants, not variables; do not try to assign them a new stream pointer value.

You can use the MS-DOS redirection symbols (> or >>) or the pipe symbol (|) to redefine the standard input and standard output for a particular program. (See your operating system manual for a complete discussion of redirection and pipes.) For example, if you execute a program and redirect

its output to a file named *results*, the program writes to the *results* file each time the standard output is specified in a write operation. Notice that you don't change the program when you redirect the output. You simply change the file associated with **stdout** for a single execution of the program.

You can redefine **stdin**, **stdout**, **stderr**, **stdaux**, or **stdprn** so that they refer to a disk file or to a device. The **freopen** routine is used for this purpose. See the **freopen** reference page in Part 2 of this manual for a description of this option.

---

*Important*

At the MS-DOS command level, **stderr** (standard error) cannot be redirected.

---

### 4.7.1.3 Controlling Stream Buffering

Files opened using the stream functions are buffered by default, except for the pre-opened streams **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. The **stderr** and **stdaux** streams are unbuffered; **stdin**, **stdout**, and **stdprn** are line-buffered, meaning that the buffer is flushed whenever a newline is encountered.

You can cause a stream to be unbuffered, or associate a buffer with an unbuffered stream, by using the **setbuf** function. Buffers allocated by the system are not accessible to the user, but buffers allocated with **setbuf** are named by the user and may be manipulated as if they were variables. Buffers must have a constant size, which is defined by the manifest constant **BUFSIZ** in *stdio.h*.

Buffers are automatically flushed when they are full (**BUFSIZ** is reached), when the associated file is closed, or when a program terminates normally. You can flush buffers at other times by using the **fflush** and **flushall** routines. **Fflush** flushes a single specified stream, while **flushall** flushes all streams that are open and buffered.

### 4.7.1.4 Closing Streams

The **fclose** and **fcloseall** functions close a stream or streams. The **fclose** routine closes a single specified stream; **fcloseall** closes all open streams except **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. If your program does not explicitly close a stream, the stream is automatically closed when the program terminates. However, it is good practice to close a stream when finished with it, as the number of streams that can be open at a given time is limited.

### 4.7.1.5 Reading and Writing Data

The stream functions allow you to transfer data in a variety of ways. You can read and write binary data (a sequence of bytes), or specify reading and writing by characters, lines, or more complicated formats. The stream functions for reading and writing data are summarized at the beginning of this section; for a full description of each function, see Part 2, "Reference," of this manual.

Reading and writing operations on streams always begin at the current position of the stream, known as the "file pointer" for the stream. The file pointer is changed to reflect the new position after a read or write operation takes place. For example, if you read a single character from a stream, the file pointer is incremented by 1 byte so that the next operation begins with the first unread character. If a stream is opened for appending, the file pointer is automatically positioned at the end of the file before each write operation.

The **feof** macro detects an end-of-file condition on a stream. Once the end-of-file indicator is set, it remains set until the file is closed or rewound, or until **clearerr** is called.

You can position the file pointer anywhere in a file by using the **fseek** function. The next operation takes place at the position you specified. The **rewind** function positions the file pointer at the beginning of the file. Use the **ftell** function to determine the current position of the file pointer.

Streams associated with a device (such as a console) do not have file pointers. Data coming from or going to a console cannot be accessed randomly. Routines that use file pointers will have undefined results if used on a stream associated with a device.

#### 4.7.1.6 Detecting Errors

When an error occurs in a stream operation, an error indicator for the stream is set. You can use the **ferror** macro to test the error indicator and determine whether an error has occurred. Once an error has occurred, the error indicator for the stream remains set until the stream is closed or rewound or until you explicitly clear the error indicator by calling **clearerr**.

### 4.7.2 Low-Level Routines

| Routine | Use |
|---|---|
| **close** | Closes a file |
| **creat** | Creates a file |
| **dup** | Creates a second *handle* for a file |
| **dup2** | Reassigns a file *handle* |
| **eof** | Tests for end-of-file |
| **lseek** | Repositions file pointer to a given location |
| **open** | Opens a file |
| **read** | Reads data from a file |
| **sopen** | Opens a file for file-sharing |
| **tell** | Gets current file pointer position |
| **write** | Writes data to a file |

Low-level input and output calls do not buffer or format data. Files opened by low-level calls are referenced via a file handle, an integer value used by the operating system to refer to the file. The **open** function is used to open files; on MS-DOS Version 3.0 or later, **sopen** may be used to open a file with file-sharing attributes.

Low-level functions, unlike the stream functions, do not require the include file *stdio.h*. However, some common constants are defined in *stdio.h;* for example, the end-of-file indicator, **EOF**, may be useful. If your program requires these constants, you must include *stdio.h*.

Declarations for the low-level functions are given in the include file *io.h*.

### 4.7.2.1 Opening a File

A file must be opened with the **open**, **sopen**, or **creat** function before input and output with the low-level functions can be performed on that file. The file can be opened for reading, writing, or both, and opened in either text or binary mode. The include file *fcntl.h* must be included when opening a file, as it contains definitions for flags used in the **open**. In some cases the file *sys\types.h* must also be included; see the reference page for **open** in Part 2 of this manual for details.

These functions return a file *handle*, to be used to refer to the file in later operations. When you call one of these functions, assign the return value to an integer variable and use that variable to refer to the opened stream.

### 4.7.2.2 Predefined Handles

When a program begins execution, five file handles, corresponding to the standard input, standard output, standard error, standard auxiliary, and standard print, are already assigned. By using the following predefined handles, a program can call low-level functions to access the standard input, standard output, standard error, standard auxiliary, and standard print streams (described with the stream functions in Section 4.7.1.2).

| Stream | Handle |
|---|---|
| stdin | 0 |
| stdout | 1 |
| stderr | 2 |
| stdaux | 3 |
| stdprn | 4 |

You can use these file handles in your program without previously opening the associated files. They are automatically opened when the program begins.

As with the stream functions, you can use redirection and pipe symbols when you execute your program to redirect the standard input and standard output. The **dup** and **dup2** functions allow you to assign multiple handles for the same file; these functions are typically used to associate the predefined file handles with different files.

---

*Important*

At the MS-DOS command level, **stderr** (standard error) cannot be redirected.

---

### 4.7.2.3 Reading and Writing Data

Two basic functions, **read** and **write**, perform input and output. As with the stream functions, reading and writing operations always begin at the current position in the file. The current position is updated each time a read or write operation takes place.

The **eof** routine can be used to test for an end-of-file condition. Low-level I/O routines set the **errno** variable when an error occurs. This means that you can use the **perror** function to print information about I/O errors.

You can position the file pointer anywhere in a file by using the **lseek** function. The next operation takes place at the position you specified. Use the **tell** function to determine the current position of the file pointer.

Devices (such as the console) do not have file pointers. The **lseek** and **tell** routines have undefined results if used on a handle associated with a device.

### 4.7.2.4 Closing files

The **close** function closes an open file. Open files are automatically closed when a program terminates. However, it is good practice to close a file when finished with it, as the number of files that can be open at a given time is limited.

## 4.7.3 Console and Port I/O Routines

| Routine | Use |
|---|---|
| **cgets** | Reads a string from the console |
| **cprintf** | Writes formatted data to the console |
| **cputs** | Writes a string to the console |
| **cscanf** | Reads formatted data from the console |
| **getch** | Reads a character from the console |
| **getche** | Reads a character from the console and echoes it |
| **inp** | Reads specified I/O port |
| **kbhit** | Checks for a keystroke at the console |
| **outp** | Writes to specified I/O port |
| **putch** | Writes a character to the console |
| **ungetch** | "Ungets" the last character read from the console so that it becomes the next character read |

The console and port I/O routines are implemented as functions and are declared in the include file *conio.h*. These functions perform reading and writing operations on your console or on the specified port. The **cgets**, **cscanf**, **getch**, **getche**, and **kbhit** routines take input from the console, while **cprintf**, **cputs**, **putch**, and **ungetch** write to the console. Redirecting the standard input or standard output from the command line causes the input or output of these functions to be redirected.

The console or port does not have to be opened or closed before I/O is performed, so there are no open or close routines in this category. The port I/O routines (**inp** and **outp**) read or write 1 byte at a time from the specified port. The console I/O routines allow reading and writing of strings (**cgets** and **cputs**), formatted data (**cscanf** and **cprintf**), and characters. Several options are available when reading and writing characters.

The **putch** routine writes a character to the console. The **getch** and **getche** routines read a character from the console; **getche** echoes the character back to the console, and **getch** does not. The **ungetch** routine "ungets" the last character read; the next read operation on the console begins with the "ungotten" character.

The **kbhit** routine determines whether a key has been struck at the console. This routine allows you to test for keyboard input before you attempt to read from the console.

---

*Note*

The console I/O routines use the corresponding low-numbered MS-DOS system calls to read and write characters. See your *Microsoft MS-DOS Programmer's Reference Manual* for details on the system calls.

---

# 4.8 Math

| Routine | Use |
|---|---|
| **acos** | Calculates arc cosine |
| **asin** | Calculates arc sine |
| **atan** | Calculates arc tangent $x$ |
| **atan2** | Calculates arc tangent $y/x$ |
| **besselt** | Calculates bessel functions |
| **cabs** | Finds absolute value of complex number |
| **ceil** | Finds integer ceiling of **double** |
| **cos** | Calculates cosine |
| **cosh** | Calculates hyperbolic cosine |
| **exp** | Calculates exponential function |
| **fabs** | Finds absolute value of **double** |
| **floor** | Finds integer floor of **double** |
| **fmod** | Finds remainder |
| **frexp** | Breaks down **double** into mantissa and exponent |
| **hypot** | Calculates hypotenuse |
| **ldexp** | Calculates $x$ times a power of 2 |
| **log** | Calculates natural logarithm |
| **log10** | Calculates base 10 logarithm |
| **matherr** | Handles math errors |
| **modf** | Breaks down **double** into integer and fractional parts |
| **pow** | Calculates a power of $x$ |
| **sin** | Calculates sine |

† Note: **bessel** does not correspond to a single function but to six functions named **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**.

| | |
|---|---|
| **sinh** | Calculates hyperbolic sine |
| **sqrt** | Finds square root |
| **tan** | Calculates tangent |
| **tanh** | Calculates hyperbolic tangent |

The math routines allow you to perform common mathematical calculations. All math routines (except **matherr**, the error-handling function) work with floating-point values and thus require floating-point support (see Section 2.10, "Floating-Point Support," in Chapter 2, "Using C Library Routines"). Function declarations for the math routines are given in the include file *math.h*.

The **matherr** routine is invoked by the math functions when errors occur. This routine is defined in the library, but can be redefined by the user if different error-handling procedures are desired. The user-defined **matherr** function, if given, must conform to the specifications given on the **matherr** reference page in Part 2 of this manual.

You are not required to supply a definition for **matherr**. If no definition is present, the default error returns for each routine are used. See the reference page for each routine in Part 2 of this manual for a description of that routine's error returns.

## 4.9 Memory Allocation

| Routine | Use |
|---|---|
| **calloc** | Allocates storage for array |
| **free** | Frees an allocated block |
| **malloc** | Allocates a block |
| **realloc** | Reallocates a block |
| **sbrk** | Resets break value |

The memory allocation routines allow you to allocate, free, and re-allocate blocks of memory. They are declared in the include file *malloc.h*.

The **calloc** and **malloc** routines allocate memory blocks. The **malloc** routine allocates a given number of bytes, while **calloc** allocates and initializes to zero an array with elements of a given size. The **realloc** routine changes the size of an allocated block. These routines all return a **char** pointer, but the space to which they point satisfies the alignment requirements of any type of object. Use a type cast on the return value to obtain the type of pointer you need.

The **free** routine de-allocates memory that was previously allocated, making it available for later allocation requests.

The **sbrk** routine is a lower-level memory allocation routine. It increases the program's break value, allowing the program to take advantage of available unallocated memory.

---

*Warning*

In general, a program that uses the **sbrk** routine should not use the other memory allocation routines, although their use is not prohibited. In particular, using **sbrk** to decrease the break value may cause later calls to the other memory allocation routines to give unpredictable results.

---

## 4.10 MS-DOS Interface

| Routine | Use |
|---|---|
| **bdos** | Invokes MS-DOS system call; uses only DX and AL registers |
| **dosexterr** | Obtains register values from MS-DOS system call 59H |
| **FP_OFF** | Returns offset portion of a **far** pointer |
| **FP_SEG** | Returns segment portion of a **far** pointer |
| **int86** | Invokes MS-DOS interrupts |
| **int86x** | Invokes MS-DOS interrupts |
| **intdos** | Invokes MS-DOS system call; uses registers other than DX and AL |
| **intdosx** | Invokes MS-DOS system call; uses registers other than DX and AL |
| **segread** | Returns current values of segment registers |

These routines provide access to MS-DOS system calls and interrupts. See your *Microsoft MS-DOS Programmer's Reference Manual* for information on system calls and interrupts.

The **FP_OFF** and **FP_SEG** routines are provided to allow the user easy access to the segment and offset portions of a **far** pointer value. **FP_OFF** and **FP_SEG** are implemented as macros and defined in *dos.h*. The remaining routines are implemented as functions and declared in *dos.h*.

The **dosexterr** function obtains and stores the register values returned by MS-DOS system call 59H (extended error handling). This function is provided for use with MS-DOS Version 3.0 or later.

The **bdos** routine is useful for invoking MS-DOS calls that use either or both of the DX (DH/DL) and AL registers for arguments. However, **bdos** should not be used to invoke system calls that return an error code in AX if the carry flag is set; the program cannot detect whether the carry flag is set, making it impossible to determine whether the value in AX is a legitimate value or an error value. In this case the **intdos** routine should be used instead, since it allows the program to detect whether the carry flag is set. The **intdos** routine can also be used to invoke MS-DOS calls that

use registers other than DX and AL.

The **intdosx** routine is similar to the **intdos** routine, but is used when ES is required by the system call, when DS must contain a value other than the default data segment (for instance, when a **far** pointer is used), or when making the system call in a large model program. When calling **intdosx**, you give an argument that specifies the segment values to be used in the call.

The **int86** routine can be used to invoke MS-DOS interrupts. The **int86x** routine is similar, but, like the **intdosx** routine, is designed to work with large model programs and far items, as described in the preceding paragraph for **intdosx**.

The **segread** routine returns the current values of the segment registers. This routine is typically used with the **intdosx** and **int86x** routines to obtain the correct segment values.

# 4.11 Process Control

| Routine | Use |
|---|---|
| abort | Aborts a process |
| execl | Executes child process with argument list |
| execle | Executes child process with argument list and given environment |
| execlp | Executes child process using PATH variable and argument list |
| execv | Executes child process with argument array |
| execve | Executes child process with argument array and given environment |
| execvp | Executes child process using PATH variable and argument array |
| exit | Terminates process |
| _exit | Terminates process without flushing buffers |
| getpid | Gets process ID number |
| signal | Handles an interrupt signal |
| spawnl | Executes child process with argument list |
| spawnle | Executes child process with argument list and given environment |
| spawnlp | Executes child process using PATH variable and argument list |
| spawnv | Executes child process with argument array |
| spawnve | Executes child process with argument array and given environment |
| spawnvp | Executes child process using PATH variable and argument array |
| system | Executes an MS-DOS command |

The term "process" refers to a program being executed by the operating system. A process consists of the program's code and data, plus information pertaining to the status of the process, such as the number of open files. Whenever you execute a program at the MS-DOS level, you start a process. In addition, you can start, stop, and manage processes from within a program by using the process control routines.

The process control routines allow you to:

1. Identify a process by a unique number (**getpid**)
2. Terminate a process (**abort**, **exit**, and **_exit**)
3. Handle an interrupt signal (**signal**)
4. Start a new process (the **exec** and **spawn** families of routines, plus the **system** routine)

All process control functions except **signal** are declared in the include file *process.h*. The **signal** function is declared in *signal.h*.

The **abort** and **_exit** functions perform an immediate exit without flushing stream buffers. The **exit** call performs an exit after flushing stream buffers.

The **system** call executes a given MS-DOS command. The **exec** and **spawn** routines start up a new process, called the "child" process. The difference between the **exec** and **spawn** routines is that the **spawn** routines are capable of returning control from the child process to its caller (the "parent" process). Both the parent process and the child process are present in memory (unless **P_OVERLAY** is specified).

In the **exec** routines, the child process overlays the parent process, so returning control to the parent process is impossible (unless an error occurs when attempting to start execution of the child process).

There are six forms each of the **spawn** and **exec** routines. The differences between the forms are summarized in Table 4.1. The function names are given in the first column. The second column specifies whether the current PATH setting is used to locate the file to be executed as the child process.

The third column describes the method for passing arguments to the child process. Passing an argument list means that the arguments to the child process are listed as separate arguments in the **exec** or **spawn** call; passing an argument array means that the arguments are stored in an array, and a pointer to the array is passed to the child process. The argument list method is typically used when the number of arguments is constant or is

known at compile time, while the argument array method is useful when the number of arguments must be determined at run time.

The last column specifies whether the child process inherits the environment settings of its parent or whether a table of environment settings can be passed to set up a different environment for the child process.

**Table 4.1**

**Forms of the spawn and exec Routines**

| Routine | Use of PATH Setting | Argument-Passing Convention | Environment |
|---|---|---|---|
| **execl, spawnl** | Does not use PATH | Argument list | Inherited from parent |
| **execle, spawnle** | Does not use PATH | Argument list | Pointer to environment table for child process passed as last argument |
| **execlp, spawnlp** | Uses PATH | Argument list | Inherited from parent |
| **execv, spawnv** | Does not use PATH | Argument array | Inherited from parent |
| **execve, spawnve** | Does not use PATH | Argument array | Pointer to environment table for child process passed as last argument |
| **execvp, spawnvp** | Uses PATH | Argument array | Inherited from parent |

## 4.12 Searching and Sorting

| Routine | Use |
|---|---|
| **bsearch** | Performs binary search |
| **qsort** | Performs quick sort |

The **bsearch** and **qsort** functions provide helpful binary search and quick sort utilities. They are declared in the include file *search.h*.

# 4.13 String Manipulation

| Routine | Use |
|---|---|
| strcat | Appends a string |
| strchr | Finds first occurrence of a given character in string |
| strcmp | Compares two strings |
| strcmpi | Compares two strings without regard to case |
| strcpy | Copies one string to another |
| strcspn | Finds first occurrence of a character from given character set in string |
| strdup | Duplicates string |
| strlen | Finds length of string |
| strlwr | Converts string to lowercase |
| strncat | Appends *n* characters of string |
| strncmp | Compares *n* characters of two strings |
| strncpy | Copies *n* characters of one string to another |
| strnset | Sets *n* characters of string to given character |
| strpbrk | Finds first occurrence of character from one string in another |
| strrchr | Finds last occurrence of given character in string |
| strrev | Reverses string |
| strset | Sets all characters of string to given character |
| strspn | Finds first substring from given character set in string |
| strtok | Finds next token in string |
| strupr | Converts string to uppercase |

The string functions are declared in the include file *string.h*. A wide variety of string functions is available in the run-time library. You can

- Perform string comparisons
- Search for individual characters or characters from a given set
- Copy strings
- Convert strings to a different case
- Set characters of the string to a given character
- Reverse the characters of strings
- Break strings into tokens

All string functions work on null-terminated character strings. When working with character arrays that do not end with a null character, you can use the buffer manipulation routines, described earlier in this chapter.

## 4.14 Time

| Routine | Use |
|---|---|
| asctime | Converts time from structure to character string |
| ctime | Converts time from long integer to character string |
| ftime | Gets current system time as structure |
| gmtime | Converts time from integer to structure |
| localtime | Converts time from integer to structure with local correction |
| time | Gets current system time as long integer |
| tzset | Sets external time variables from environment time variable |
| utime | Sets file modification time |

The time functions allow you to obtain the current time, then convert and store it according to your particular needs. The current time is always taken from the system time. The **time** and **ftime** functions return the current time as the number of seconds elapsed since Greenwich Mean Time, January 1, 1970. This value can be converted, adjusted, and stored in a variety of ways, using the **asctime**, **ctime**, **gmtime**, and **localtime** functions. The **utime** function sets the modification time for a specified file,

using either the current time or a time value stored in a structure.

The **ftime** function requires two include files: *sys\types.h* and *sys\timeb.h*. The **ftime** function is declared in *sys\timeb.h*. The **utime** function also requires two include files: *sys\types.h* and *sys\utime.h*. The **utime** function is declared in *sys\utime.h*. The remainder of the time functions are declared in the include file *time.h*.

When you want to use **ftime** or **localtime** to make adjustments for local time, you must define an environment variable named TZ. See Section 3.2 on the global variables **daylight**, **timezone**, and **tzname** for a discussion of the TZ variable; TZ is also described on the **tzset** reference page in Part 2 of this manual.

## 4.15 Miscellaneous

| Routine | Use |
|---|---|
| **abs** | Finds absolute value of **int** |
| **assert** | Tests for logic error |
| **getenv** | Gets value of environment variable |
| **labs** | Finds absolute value of **long** |
| **longjmp** | Restores a saved stack environment |
| **perror** | Prints error message |
| **putenv** | Adds or modifies value of environment variable |
| **rand** | Gets a pseudo-random number |
| **setjmp** | Saves a stack environment |
| **srand** | Initializes pseudo-random series |
| **swab** | Swaps bytes of data |

The "miscellaneous" category covers a number of commonly-used routines that do not fit easily into any of the other categories. All routines except **assert**, **longjmp**, and **setjmp** are declared in *stdlib.h*. The **assert** routine is a macro and is defined in *assert.h*. The **setjmp.h** and **longjmp.h** functions are declared in *setjmp.h*.

The **abs** and **labs** functions return the absolute value of an **int** and a **long** value, respectively. (A macro named **abs** is available in the include file *v2tov3.h*; the macro gives the absolute value for any type.)

The **assert** macro is typically used to test for program logic errors; it prints a message when a given "assertion" fails to hold true. Defining the identifier **NDEBUG** to any value causes occurrences of **assert** to be removed from the source file, thus allowing you to turn off assertion-checking without modifying the source file.

The **getenv** and **putenv** routines provide access to the environment table. The global variable **environ** also points to the environment table, but it is recommended that you use **getenv** and **putenv** routines to access and modify environment settings rather than accessing the environment table directly.

The **perror** routine prints the system error message, along with a user-supplied message, for the last system-level call that produced an error. **Perror** is declared in *stdlib.h*. The error number is obtained from the **errno** variable. The system message is taken from the **sys_errlist** array. The **errno** variable is only guaranteed to be set upon error for those routines that explicitly mention the **errno** variable in the "Return Value" section of the reference pages in Part 2 of this manual.

The **rand** and **srand** functions initialize and generate a pseudo-random sequence of integers.

The **setjmp** and **longjmp** functions save and restore a stack environment. These routines let you execute a nonlocal goto.

The **swab** routine (also declared in *stdlib.h*) swaps bytes of binary data. It is typically used to prepare data for transfer to a machine that uses a different byte order.

# Chapter 5
# Include Files


5.1 Introduction 69
5.2 assert.h 70
5.3 conio.h 70
5.4 ctype.h 70
5.5 direct.h 71
5.6 dos.h 71
5.7 errno.h 72
5.8 fcntl.h 72
5.9 io.h 73
5.10 malloc.h 73
5.11 math.h 73
5.12 memory.h 74
5.13 process.h 74
5.14 search.h 75
5.15 setjmp.h 75
5.16 share.h 75
5.17 signal.h 75
5.18 stdio.h 76
5.19 stdlib.h 77
5.20 string.h 78
5.21 sys\locking.h 78

5.22 sys\stat.h 78
5.23 sys\timeb.h 79
5.24 sys\types.h 79
5.25 sys\utime.h 79
5.26 time.h 79
5.27 v2tov3.h 80

# 5.1 Introduction

The include files provided with the run-time library contain macro and constant definitions, type definitions, and function declarations. Some routines require definitions and declarations from include files to work properly; for other routines, the inclusion of a file is optional. The description of each include file in this chapter explains the contents of each include file and lists the routines that use it.

Two sets of function declarations are provided in each include file. The first set declares both the function return type and the argument type list for the function. This set is included only when you enable argument type-checking by defining **LINT_ARGS**, as described in Section 2.5, "Argument Type-Checking," of Chapter 2, "Using C Library Routines." The second set of declarations declares only the function return type. This set is included when argument type-checking is *not* enabled.

The include files were named and organized to meet the following objectives.

- To maintain compatibility with the names of include files on XENIX and UNIX systems
- To reflect the logical categories of run-time routines (for example, placing declarations for all memory allocation functions in one file, *malloc.h*)
- To require the inclusion of the minimum number of files to use a given routine

Occasionally these goals conflict. For example, the **ftime** function uses the structure type **timeb**. The **timeb** structure type is defined in the include file *sys\timeb.h* on XENIX systems; to maintain compatibility, the same include file is used on MS-DOS. To minimize the number of required include files when using **ftime**, the **ftime** function is declared in *sys\timeb.h*, even though most of the other time functions are declared in *time.h*.

## 5.2 assert.h

The include file *assert.h* defines the **assert** macro. The *assert.h* file must be included when **assert** is used.

The definition of **assert** is enclosed in an **#ifndef** preprocessor block. If the identifier **NDEBUG** has not been defined (through a **#define** directive or on the compiler command line), the **assert** macro is defined to test a given expression (the "assertion"); if the assertion is false, a message is printed and the program is terminated.

If **NDEBUG** is defined, however, **assert** is defined as empty text. This disables all program assertions by removing all occurrences of **assert** from the source file. Thus, you can suppress program assertions by defining **NDEBUG**.

## 5.3 conio.h

The *conio.h* include file contains function declarations for all of the console and port I/O routines, as listed below.

| | | | |
|---|---|---|---|
| cgets | cscanf | inp | putch |
| cprintf | getch | kbhit | ungetch |
| cputs | getche | outp | |

## 5.4 ctype.h

The *ctype.h* include file defines macros and constants and declares a global variable used in character classification. The macros defined in *ctype.h* are listed below; you must include *ctype.h* when using these macros or the macros will be undefined.

| | | | | | |
|---|---|---|---|---|---|
| isalnum | iscntrl | islower | isspace | toascii | _tolower |
| isalpha | isdigit | isprint | isupper | tolower | _toupper |
| isascii | isgraph | ispunct | isxdigit | toupper | |

The **toupper** and **tolower** macros are defined as conditional operations. These macros evaluate their argument twice, and so produce unexpected results for arguments with side effects. To overcome this problem, you can remove the macro definitions of **toupper** and **tolower** and use the functions by the same names; see Section 4.3, "Character Classification and Conversion," in Chapter 4, "Run-Time Routines by Category," for details. Declarations for the function versions of **tolower** and **toupper** are given in *stdlib.h*.

In addition to macro definitions, the *ctype.h* include file also contains the following.

1. A set of manifest constants defined as bit masks. The bit masks correspond to specific classification tests. For example, the constants **_UPPER** and **_LOWER** are defined to test for an uppercase or lowercase letter, respectively.
2. A declaration of a global variable, **_ctype**. The **_ctype** variable is a table of character classification codes based on ASCII character codes.

## 5.5 direct.h

The *direct.h* include file contains function declarations for the four directory control functions (**chdir**, **getcwd**, **mkdir**, and **rmdir**).

## 5.6 dos.h

The *dos.h* include file contains macro definitions, function declarations, and type definitions for the MS-DOS interface functions.

The **FP_SEG** and **FP_OFF** macros are defined to obtain the segment and offset portions from a **far** pointer value. You must include *dos.h* when using these macros or they will be undefined.

The following functions are declared in *dos.h*.

| | | | |
|---|---|---|---|
| **bdos** | **int86** | **intdos** | **segread** |
| **dosexterr** | **int86x** | **intdosx** | |

The *dos.h* file also defines the **WORDREGS** and **BYTEREGS** structure types, used to define sets of word registers and byte registers, respectively. These structure types are combined in the **REGS** union type. The **REGS** union serves as a general purpose register type, holding both register structures at one time. The **SREGS** structure type defines four members to hold the ES, CS, SS, and DS segment register values.

The **DOSERROR** structure is defined to hold error values returned by MS-DOS system call 59H (available under MS-DOS 3.0 and later).

Notice that **WORDREGS**, **BYTEREGS**, **REGS**, **SREGS**, and **DOSERROR** are tags, not **typedef** names. (See the *Microsoft C Language Reference* for a discussion of type definitions, tags, and **typedef** names.)

## 5.7 errno.h

The **errno.h** include file defines the values used by system-level calls to set the **errno** variable. The constants defined in **errno.h** are used by the **perror** function to index the corresponding error message in the global variable **sys_errlist**.

The constants defined in **errno.h** are listed with the corresponding error messages in Appendix A, "Error Messages."

## 5.8 fcntl.h

The include file *fcntl.h* defines flags used in the **open** and **sopen** calls to specify the type of operations for which the file is opened and to control whether the file is interpreted in text or binary mode. This file should always be included when **open** or **sopen** is used.

The function declarations for **open** and **sopen** are not in *fcntl.h*; instead, they are given in the include file *io.h*.

## 5.9 io.h

The include file *io.h* contains function declarations for most of the file handling and low-level I/O functions, as listed below.

| | | | |
|---|---|---|---|
| access | dup2 | mktemp | tell |
| chmod | eof | open | umask |
| chsize | filelength | read | unlink |
| close | isatty | rename | write |
| creat | locking | setmode | |
| dup | lseek | sopen | |

The exceptions are **fstat** and **stat**, which are declared in *sys\stat.h*.

## 5.10 malloc.h

The include file *malloc.h* contains function declarations for the five memory allocation functions **calloc**, **free**, **malloc**, **realloc**, and **sbrk**.

## 5.11 math.h

The include file *math.h* contains function declarations for all floating-point math routines, plus the **atof** routine, as listed below.

| | | | | |
|---|---|---|---|---|
| acos | cabs | double | log10 | sqrt |
| asin | ceil | floor | matherr | tan |
| atan | cos | fmod | modf | tanh |
| atan2 | cosh | hypot | pow | |
| atof | exp | idexp | sin | |
| bessel† | fabs | log | sinh | |

The *math.h* include file also defines two structures, **exception** and **complex**. The **exception** structure is used with the **matherr** function, and the **complex** structure is used to declare the argument to the **cabs** function.

---

† Note: **bessel** does not correspond to a single function but to six functions named j0, j1, jn, y0, y1, and yn.

The **HUGE** value, which is returned on error from some math routines, is defined in *math.h*. The **HUGE** value may be implemented either as a manifest constant or as a global variable with **double** type. The value of the **HUGE** constant must not be changed.

The *math.h* file also defines manifest constants passed in the **exception** structure when a math routine generates an error (for example, **DOMAIN** and **SING**).

## 5.12 memory.h

The include file *memory.h* contains function declarations for the six buffer manipulation routines: **memccpy**, **memchr**, **memcmp**, **memcpy**, **memset**, and **movedata**.

## 5.13 process.h

The include file *process.h* declares all process control functions (listed below) except for the **signal** function, which is declared in *signal.h*.

| | | | | | |
|---|---|---|---|---|---|
| **abort** | **execlp** | **execvp** | **getpid** | **spawnlp** | **spawnvp** |
| **execl** | **execv** | **exit** | **spawnl** | **spawnv** | **system** |
| **execle** | **execve** | **_exit** | **spawnle** | **spawnve** | |

The *process.h* include file also defines flags used in calls to **spawn** functions to control execution of the child process. Whenever you use one of the six **spawn** functions, you must include *process.h* so the flags are defined.

## 5.14 search.h

The include file *search.h* declares the two functions **bsearch** and **qsort**.

## 5.15 setjmp.h

The include file *setjmp.h* contains function declarations for the **setjmp** and **longjmp** functions. It also defines the machine-dependent buffer used by the **setjmp** and **longjmp** functions to save and restore the program state.

## 5.16 share.h

The include file *share.h* defines flags used in the **sopen** function to set the sharing mode of a file. This file should be included whenever **sopen** is used. The function declaration for **sopen** is given in the file *io.h*. Note that the **sopen** function should only be used under MS-DOS Version 3.0 or later.

## 5.17 signal.h

The include file *signal.h* defines the values for signals. Only the **SIGINT** signal (CONTROL-C) is recognized on MS-DOS. The **signal** function is also declared in *signal.h*.

# 5.18 stdio.h

The include file *stdio.h* contains definitions of constants, macros, and types, along with function declarations for stream I/O routines. The stream I/O routines are listed below.

| | | | | |
|---|---|---|---|---|
| clearerr | fgetchar | fread | gets | scanf |
| fclose | filenot | freopen | getw | setbuf |
| fcloseall | flushall | fscanf | printf | sprintf |
| fdopen | fopen | fseek | putct | sscanf |
| feoft | fprintf | ftell | putchart | ungetc |
| ferrort | fputc | fwrite | puts | |
| fflush | fputchar | getct | putw | |
| fgetc | fputs | getchart | rewind | |

The *stdio.h* file also defines the constants listed below. You can use these constants in your programs, but you should not alter their values.

**BUFSIZ** Buffers used in stream I/O are required to have a constant size, which is defined by the **BUFSIZ** constant. This value is used to establish the size of system-allocated buffers, and must also be used when calling **setbuf** to allocate your own buffers.

**_NFILE** The **_NFILE** constant defines the number of open files allowed at one time. The five files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** are always open, so you should include them when calculating the number of files your program opens.

**EOF** The **EOF** value is defined to be the value returned by an I/O routine when the end of the file (or in some cases, an error) is encountered.

**NULL** The **NULL** value is the null pointer value. It is defined as 0 in small and medium model programs and as 0L in large model programs.

The *stdio.h* file also defines a number of flags used internally to control stream operations.

---

† Implemented as a macro.

The FILE structure type is defined in *stdio.h*. Stream routines use a pointer to the FILE type to access a given stream. The system uses the information in the FILE structure to maintain the stream.

The FILE structures are stored as an array called _iob, with one entry per file. Thus, each element of _iob is a FILE structure corresponding to a stream. When a stream is opened, it is assigned the address of an entry in the _iob array (a FILE pointer). Thereafter, the pointer is used for references to the stream.

## 5.19 stdlib.h

The *stdlib.h* include file contains function declarations for the following routines.

| | | | | |
|---|---|---|---|---|
| abs | fcvt | labs | rand | toupper |
| atoi | gcvt | ltoa | srand | ultoa |
| atol | getenv | perror | swab | |
| ecvt | itoa | putenv | tolower | |

The tolower and toupper routines are functions in the run-time library, but they are also implemented as macros in the include file *ctype.h*. The declarations for tolower and toupper are enclosed in an #ifndef block; they take effect only if the corresponding macro definitions in *ctype.h* have been suppressed by removing the definitions of tolower and toupper. For instructions on using these routines as macros or as functions, see Section 4.3, "Character Classification and Conversion," in Chapter 4, "Run-Time Routines by Category."

*Stdlib.h* also includes declarations of the following global variables.

| | | |
|---|---|---|
| _doserrno | _fmode | _psp |
| environ | _osmajor | sys_errlist |
| errno | _osminor | sys_nerr |

## 5.20 string.h

The *string.h* include file declares the string manipulation functions, as listed below.

| | | | | |
|---|---|---|---|---|
| **strcat** | **strcpy** | **strlwr** | **strnset** | **strset** |
| **strchr** | **strcspn** | **strncat** | **strpbrk** | **strspn** |
| **strcmp** | **strdup** | **strncmp** | **strrchr** | **strtok** |
| **strcmpi** | **strlen** | **strncpy** | **strrev** | **strupr** |

## 5.21 sys\locking.h

The *locking.h* include file (conventionally stored in a subdirectory named SYS) contains definitions of flags used in calls to **locking**. Whenever you use the **locking** routine, you must include this file so that the locking flags are defined.

The function declaration for **locking** is given in the file *io.h*. Note that the **locking** function should only be used under MS-DOS Version 3.0 or later.

## 5.22 sys\stat.h

The *stat.h* include file (conventionally stored in a subdirectory named SYS) defines the structure type returned by the **fstat** and **stat** functions and defines flags used to maintain file status information. It also contains function declarations for the **fstat** and **stat** functions. Whenever you use the **fstat** or **stat** function, you must include this file so that the appropriate structure type (named **stat**) is defined.

## 5.23 sys\timeb.h

The include file *timeb.h* (conventionally stored in a subdirectory named SYS) defines the **timeb** structure type and declares the **ftime** function, which uses the **timeb** structure type. Whenever you use the **ftime** function you must include *timeb.h* so that the structure type is defined.

## 5.24 sys\types.h

The include file *types.h* (conventionally stored in a subdirectory named SYS) defines types used by system-level calls to return file status and time information. You must include this file whenever the file *sys\stat.h*, *sys\utime.h*, or *sys\timeb.h* is included.

## 5.25 sys\utime.h

The include file *utime.h* (conventionally stored in a subdirectory named SYS) defines the **utimbuf** structure type and declares the **utime** function, which uses the **utimbuf** type. Whenever you use the **utime** function you must include *utime.h* so that the structure type is defined.

## 5.26 time.h

The *time.h* include file declares the time functions **asctime**, **ctime**, **gmtime**, **localtime**, **time**, and **tzset**. (The **ftime** and **utime** functions are declared in *sys\timeb.h* and *sys\utime.h*, respectively.)

*Time.h* also defines the **tm** structure, used by the **asctime**, **gmtime**, and **localtime** functions.

# 5.27 v2tov3.h

The include file *v2tov3.h* is provided for users who are converting from Version 2.03 or earlier of the Microsoft C Compiler. Some of the routines provided in the Version 2.03 run-time library are supported in a slightly different form under Version 3.0 of the compiler. Including *v2tov3.h* allows those routines to be used in their original form without altering the source code.

The *v2tov3.h* file, along with other differences between Version 3.0 of the Microsoft C Compiler and other versions, is discussed in detail in Appendix D, "Converting from Previous Versions of the Compiler," in the *Microsoft C Compiler User's Guide.*

The *v2tov3.h* file contains 3 macro definitions that may be useful. The **abs** macro produces the absolute value of its argument. The **min** and **max** macros calculate the minimum and maximum, respectively, of two numbers. See the *v2tov3.h* include file for details.

Library Reference

Reference

# Part 2
# Reference

## Summary

```
#include <process.h>     /* required only for function declarations */
void abort( );
```

## Description

The **abort** function prints the message

`Abnormal program termination`

to **stderr**, then terminates the calling process, returning control to the process that initiated the calling process (usually the operating system). **Abort** does not flush stream buffers.

## Return Value

An exit status of 3 is returned to the parent process or operating system.

## See Also

**execl, execle, execlp, execv, execve, execvp, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp, signal**

## Example

```
#include <stdio.h>
#include <process.h>

FILE *stream;

if ((stream = fopen("data","r")) == NULL) {
        perror("couldn't open data file");
        abort();
        }
```

## abs

### Summary

```
#include <stdlib.h>    /* required only for function declarations */

int abs(n);
int n;                 /* integer value */
```

### Description

The **abs** function returns the absolute value of its integer argument *n*.

### Return Value

**Abs** returns the absolute value of its argument. There is no error return.

### See Also

**cabs, fabs, labs**

### Example

```
#include <stdlib.h>

int x = -4, y;

y = abs(x);                    /* y = 4 */
```

## Summary

```
#include <io.h>                    /* required only for function declarations */

int access(pathname, mode);
char *pathname ;                   /* file pathname */
int mode;                          /* permission setting */
```

## Description

The **access** function determines whether the specified file exists and can be accessed in the given *mode*. The possible values for *mode* and their meanings in the **access** call are

| Value | Meaning |
|---|---|
| 06 | Check for read and write permission |
| 04 | Check for read permission |
| 02 | Check for write permission |
| 00 | Check for existence only |

Under MS-DOS all existing files have read access; thus, the modes 00 and 04 produce the same result. Similarly, the modes 06 and 02 are equivalent, since write access implies read access on MS-DOS.

## Return Value

**Access** returns the value 0 if the file has the given *mode*. A return value of −1 indicates that the named file does not exist or is not accessible in the given *mode*, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Access denied: the file's permission setting does not allow the specified access. |
| ENOENT | File or pathname not found. |

## See Also

chmod, fstat, open, stat

## Example

```
#include <io.h>
#include <fcntl.h>

int fh;
        .
        .
        .
            /* check for write permission */
if ((access("data",2)) == -1) {
            perror("data file not writable");
            exit(1);
            }

fh = open("data",O_WRONLY);
```

## Summary

```
#include <math.h>

double acos(x);
double x;
```

## Description

The **acos** function returns the arc cosine of $x$ in the range 0 to $\pi$. The value of $x$ must be between −1 and 1.

## Return Value

**Acos** returns the arc cosine result. If $x$ is less than −1 or greater than 1, **acos** sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0.

Error handling can be modified by using the **matherr** routine.

## See Also

**asin, atan, atan2, cos, matherr, sin, tan**

## Example

```
#include <math.h>
#include <stdio.h>

double x, y;
     .
     .
     .
if (x > 1.0)
        printf("Error: %f too large for acos\n", x);
else if (x < -1.0)
        printf("Error: %f too small for acos\n", x);
else
        y = acos(x);
```

This example prints an error message if $x$ is greater than 1 or less than −1; otherwise, the arc cosine of $x$ is assigned to $y$.

# asctime

## Summary

```
#include <time.h>

char *asctime(time);
struct tm *time;                /* pointer to structure defined in time.h */
```

## Description

The **asctime** function converts a time stored as a structure to a character string. The *time* value is usually obtained from a call to **gmtime** or **localtime**, both of which return a pointer to a **tm** structure, defined in *time.h*. (See **gmtime** for a description of the **tm** structure fields).

The string result produced by **asctime** contains exactly 26 characters and has the form of the following example:

```
Mon Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The newline character ('\n') and the null character ('\0') occupy the last two positions of the string.

## Return Value

**Asctime** returns a pointer to the character string result. There is no error return.

## See Also

ctime, ftime, gmtime, localtime, time, tzset

---

*Note*

The **asctime** and **ctime** functions use a single statically allocated buffer to hold the return string. Each call to one of these routines destroys the result of the previous call.

---

## Example

```
#include <time.h>
#include <stdio.h>

struct tm *newtime;
long ltime;
   .
   .
   .
time(&ltime);                          /* get time in seconds */
newtime = localtime(&ltime);           /* convert to struct tm */
                                       /* print local time
                                       ** as string
                                       */
printf("the current date and time are %s\n",
        asctime(newtime));
```

# asin

## Summary

```
#include <math.h>

double asin(x);
double x;
```

## Description

The **asin** function calculates the arc sine of $x$ in the range $-\pi/2$ to $\pi/2$. The value of $x$ must be between $-1$ and 1.

## Return Value

**Asin** returns the arc sine result. If $x$ is less than $-1$ or greater than 1, **asin** sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0.

Error handling can be modified by using the **matherr** routine.

## See Also

**acos, atan, atan2, cos, matherr, sin, tan**

## Example

```
#include <math.h>
#include <stdio.h>

double x, y;
     .
     .
     .
if (x > 1.0)
        printf("Error: %f too large for asin\n",x);
else if (x < -1.0)
        printf("Error: %f too small for asin\n", x);
else
        y = asin(x);
```

This example prints an error message if $x$ is greater than 1 or less than $-1$; otherwise, the arc sine of $x$ is assigned to $y$.

## Summary

#include <assert.h>

void **assert**(*expression*);

## Description

The **assert** routine prints a diagnostic message and terminates the calling process if *expression* is false (zero). The diagnostic message has the following form.

`Assertion failed:  file` *filename*`, line` *linenumber*

*Filename* is the name of the source file and *linenumber* is the line number of the assertion which failed in the source file. No action is taken if *expression* is true (nonzero).

The **assert** routine is typically used to identify program logic errors. The given *expression* should be chosen so that it holds true only if the program is operating as intended. After a program has been debugged, the special "no debug" identifier **NDEBUG** can be used to remove **assert** calls from the program. If **NDEBUG** is defined (to any value) with a /D command line option or with a **#define** directive, the C preprocessor removes all **assert** calls from the program source.

## Return Value

There is no return value.

---

*Note*

**Assert** is implemented as a macro.

---

### Example

```
#include <stdio.h>
#include <assert.h>

analyze_string (string, length)
char *string;
int length;
{
        assert(string != NULL);  /* can't be NULL */
        assert(*string != '\0'); /* can't be empty */
        assert(length > 0);      /* must be positive */
           .
           .
           .
}
```

In this example, the **assert** routine is used to test arguments before processing them.

## Summary

```
#include <math.h>

double atan(x);          /* calculate arc tangent of x */
double x;

double atan2(x, y);      /* calculate arc tangent of y/x */
double x;
double y;
```

## Description

The **atan** and **atan2** functions calculate the arc tangent of $x$ and $y / x$, respectively: **atan** returns a value in the range $-\pi/2$ to $\pi/2$; **atan2** returns a value in the range $-\pi$ to $\pi$.

## Return Value

Both **atan** and **atan2** return the arc tangent result. If both arguments of **atan2** are zero, the function sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns 0.

Error handling can be modified by using the **matherr** routine.

## See Also

**acos, asin, cos, matherr, sin, tan**

## Example

```
#include <math.h>

double y;

y = atan(1.0);              /* y = pi/4 */
y = atan2(1.0,-1.0);        /* y = -pi/4 */
```

This example shows calls to **atan** and **atan2**; for each call, the result is assigned to *y*.

## Summary

```
#include <math.h>

double atof(string);          /* convert string to double */
char *string;                 /* string to be converted */

#include <stdlib.h>           /* required only for function declarations */

int atoi(string);             /* convert string to int */
long atol(string);            /* convert string to long */
char *string;                 /* string to be converted */
```

## Description

These functions convert a character string to a double-precision floating-point value (**atof**), an integer value (**atoi**), or a long integer value (**atol**). The input *string* is a sequence of characters that can be interpreted as a numerical value of the specified type. The function stops reading the input string at the first character it cannot recognize as part of a number (which may be the null character terminating the string).

**Atof** expects *string* to have the following form.

[*whitespace*] [*sign*] [*digits*] [.*digits*] [e [*sign*] *digits*]

*Whitespace* consists of space and/or tab characters, which are ignored. *Sign* is either "+" or "–". *Digits* are one or more decimal digits; if no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits may be followed by an exponent, introduced by the letter "e" or "E" and consisting of a possibly signed decimal integer.

The **atoi** and **atol** functions do not recognize decimal points or exponents. The *string* argument for these functions has the form

[*whitespace*] [*sign*] *digits*

where *whitespace*, *sign*, and *digits* are exactly as described above for **atof**.

## Return Value

Each function returns the **double**, **int**, or **long** value produced by interpreting the input characters as a number. The return value is 0 (0L for **atol**) if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.

## See Also

**ecvt, fcvt, gcvt**

## Example

```
#include <stdlib.h>
#include <math.h>

double x;
int i;
long l;
char *s;

s = "  -2309.12E-15";
x = atof(s);                    /* x = -2309.12E-15 */

s = "  -9885";
i = atoi(s);                    /* i = -9885 */

s = "98854 dollars";
l = atol(s);                    /* l = 98854 */
```

The above examples show how numbers stored as strings can be converted to numerical values using the **atof**, **atoi**, and **atol** functions.

## Summary

```
#include <dos.h>

int bdos(dosfn, dosdx, dosal);
int dosfn;                      /* function number */
unsigned int dosdx;             /* DX register value */
unsigned int dosal;             /* AL register value */
```

## Description

The **bdos** function invokes the MS-DOS system call specified by *dosfn* after placing the values specified by *dosdx* and *dosal* in the DX and AL registers, respectively. **Bdos** executes an INT 21H instruction to invoke the system call. When the system call returns, **bdos** returns the content of the AX register.

**Bdos** is intended to be used to invoke DOS system calls that either take no arguments or only take arguments in the DX (DH,DL) and/or AL registers.

## Return Value

**Bdos** returns the value of the AX register after the system call has completed.

## See Also

**intdos, intdosx**

---

*Warning*

This call should *not* be used to invoke system calls that indicate errors by setting the carry flag. Since C programs do not have access to this flag, the status of the return value cannot be determined. The **intdos** function should be used in these cases.

---

## Example

```
#include <bdos.h>

char *buffer = "Enter file name:$";

        /* AL is not needed, so 0 is used */
bdos(9,(unsigned)buffer,0);
```

The example makes MS-DOS function call 9 (display string) to display a prompt. Since the AL register value is not needed, 0 is used. This example works correctly only in small and medium model programs.

## Summary

```
#include <math.h>
double j0(x);
double j1(x);
double jn(n,x);
double y0(x);
double y1(x);
double yn(n,x);

double x;          /* floating-point value */
int n;             /* integer order */
```

## Description

The **j0**, **j1**, and **jn** routines return Bessel functions of the first kind of orders 0, 1, and $n$, respectively.

The **y0**, **y1**, and **yn** routines return Bessel functions of the second kind of orders 0, 1, and $n$, respectively. The argument $x$ must be positive.

## Return Value

These functions return the result of a Bessel function of $x$.

For **j0**, **j1**, **y0**, or **y1**, if $x$ is too large, the routine sets **errno** to **ERANGE**, prints a **TLOSS** error message to **stderr**, and returns 0.

For **y0**, **y1**, or **yn**, if $x$ is negative, the routine sets **errno** to **EDOM**, prints a **DOMAIN** error message to **stderr**, and returns the value **HUGE**.

Error handling can be modified by using the **matherr** routine.

## See Also

matherr

## Example

```
#include <math.h>

double x, y, z;
     .
     .
     .
y = j0(x);
z = yn(3,x);
```

## Summary

```
#include <search.h>          /* required only for function declarations */

char *bsearch(key, base, num, width, compare);
char *key;                   /* search key */
char *base;                  /* pointer to base of search data */
unsigned num, width;         /* number and width of elements */
int (*compare)( );           /* pointer to compare function */
```

## Description

The **bsearch** function performs a binary search of a sorted array of *num* elements, each of *width* bytes in size. *Base* is a pointer to the base of the array to be searched, and *key* is the value being sought.

*Compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **Bsearch** will call the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The routine must compare the elements, then return one of the following values.

| Value | Meaning |
|---|---|
| Less than 0 | *element1* less than *element2* |
| 0 | *element1* identical to *element2* |
| Greater than 0 | *element1* greater than *element2* |

## Return Value

**Bsearch** returns a pointer to the first occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns **NULL**.

## See Also

**qsort**

## Example

```
#include <search.h>

int compare();   /* must declare as a function */

main (argc, argv)
int argc;
char **argv;
{
        char **result;
        char *key = "PATH";
           .
           .
           .
/* The following statement finds the argument that
** starts with "PATH", assuming the argument array has been
** sorted (see the qsort reference page for a sorting example).
*/
        result = (char **) bsearch(&key,argv,argc,
                sizeof(char *),compare);
           .
           .
           .
}

int compare (arg1, arg2)
char **arg1, **arg2;
{
        return (strncmp(*arg1,*arg2,strlen(*arg1)));
}
```

## Summary

```
#include <math.h>

double cabs(z);
struct complex z;      /* contains real and imaginary parts */
```

## Description

The **cabs** function calculates the absolute value of a complex number. The complex number must be a structure with type **complex**, defined in *math.h* as follows.

```
struct complex {
        double x,y;
        };
```

A call to **cabs** is equivalent to

```
sqrt(z.x * z.x + z.y * z.y)
```

## Return Value

**Cabs** returns the absolute value as described above. There is no error return.

## See Also

**abs, fabs, labs**

## Example

```
#include <math.h>

struct complex value;
double d;

value.x = 3.0;
value.y = 4.0;

d = cabs(value);
```

# calloc

## Summary

```
#include <malloc.h>  /* required only for function declarations */

char *calloc(n, size  );
unsigned n;          /* number of elements */
unsigned size;       /* length in bytes of each element */
```

## Description

The **calloc** function allocates storage space for an array of *n* elements, each of length *size* bytes. Each element is initialized to 0.

## Return Value

**Calloc** returns a **char** pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **char**, use a type cast on the return value. The return value is **NULL** if there is insufficient memory available.

## See Also

**free, malloc, realloc**

## Example

```
#include <malloc.h>

long *lalloc;
    .
    .
    .
/* Allocate enough space for 40 long integers and
** initialize it to zero.
*/
lalloc = (long *)calloc(40,sizeof(long));
```

## Summary

```
#include <math.h>

double ceil(x);
double x;                    /* floating-point value */
```

## Description

The **ceil** function returns a **double** value representing the smallest integer that is greater than or equal to $x$.

## Return Value

**Ceil** returns the **double** result. There is no error return.

## See Also

**floor, fmod**

## Example

```
#include <math.h>
double y;
     .
     .
     .
y = ceil(1.05);              /* y = 2.0 */
y = ceil(-1.05);             /* y = -1.0 */
```

# cgets

## Summary

```
#include <conio.h>    /* required only for function declarations */

char *cgets(str);
char *str;            /* storage location for data */
```

## Description

The **cgets** function reads a string of characters directly from the console and stores the string and its length in the location pointed to by *str*. The *str* must be a pointer to a character array. The first element of the array, *str*[0], must contain the maximum length (in characters) of the string to be read. The array must have enough elements to hold the string, a terminating null character ('\0'), and two additional bytes.

**Cgets** continues to read characters until a carriage return/linefeed combination (CR-LF) is read, or the specified number of characters have been read. The string is stored starting at *str*[2]. If a CR-LF combination is read, it is replaced with a null character ('\0') before being stored. **Cgets** then stores the actual length of the string in the second array element, *str*[1].

## Return Value

**Cgets** returns a pointer to the start of the string, which is at *str*[2]. There is no error return.

## See Also

getch, getche

## Example

```
#include <conio.h>

char buffer[82];
char *result;
int numread;
   .
   .
   .
*buffer = 80;    /* maximum number of characters */
                 /* note that *buffer is equivalent
                 ** to *buffer[0]
                 */

/* The following statements input a string from the
** keyboard and find its length.
*/

result = cgets(buffer);
numread = buffer[1];

/* Result points to the string, and numread is its
** length (not counting the carriage return, which has
** been replaced by a null character).
*/
```

# chdir

## Summary

```
#include <direct.h>          /* required only for function declarations */

int chdir(pathname);
char *pathname;              /* pathname of new working directory */
```

## Description

The **chdir** function causes the current working directory to be changed to the directory specified by *pathname*. *Pathname* must refer to an existing directory.

## Return Value

**Chdir** returns a value of 0 if the working directory is successfully changed. A return value of −1 indicates an error; in this case **errno** is set to **ENOENT**, indicating that the specified pathname could not be found. No error occurs if *pathname* specifies the current working directory.

## See Also

**mkdir, rmdir, system**

## Example

```
#include <direct.h>

/* The following statement changes the current working
** directory to the root directory.
*/

chdir("/");       /* Note:  equivalent to chdir("\\") */
```

## Summary

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>                    /* required only for function declarations */

int chmod(pathname, pmode);
char *pathname;                    /* pathname of existing file */
int pmode;                         /* permission setting for file */
```

## Description

The **chmod** function changes the permission setting of the file specified *pathname*. The permission setting controls read and write access to the file. *Pmode* is a constant expression containing one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in *sys\stat.h*. Any other values for *pmode* are ignored. When both constants are given, they are joined with the bitwise OR operator (|). The meaning of the *pmode* argument is as follows.

| Value | Meaning |
|---|---|
| **S_IWRITE** | Writing permitted |
| **S_IREAD** | Reading permitted |
| **S_IREAD \| S_IWRITE** | Reading and writing permitted |

If write permission is not given, the file is made read-only. Under MS-DOS all files are readable; it is not possible to give write-only permission. Thus, the modes **S_IWRITE** and **S_IREAD | S_IWRITE** are equivalent.

## Return Value

**Chmod** returns the value 0 if the permission setting is successfully changed. A return value of −1 indicates an error; in this case **errno** is set to **ENOENT**, indicating that the specified file could not be found.

## See Also

access, creat, fstat, open, stat

## Example

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>

int result;
   .
   .
   .
result = chmod("data",S_IREAD);  /* make file read-only */
if (result == -1)
        perror("can't change file mode");
```

## Summary

```
#include <io.h>                 /* required only for function declarations */

int chsize(handle, size);
int handle;                     /* handle referring to open file */
long size;                      /* new length of file in bytes */
```

## Description

The **chsize** function extends or truncates the file associated with *handle* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters ('\0') are appended if the file is extended. If the file is truncated, all data from the end of the shortened file up to the original length of the file are lost.

## Return Value

**Chsize** returns the value 0 if the file size is successfully changed. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | The specified file is read-only. Under MS-DOS 3.0 and later, **EACCES** may indicate a locking violation (the specified file is locked against access). |
| EBADF | Invalid file handle. |
| ENOSPC | No space left on device. |

## See Also

close, creat, open

## Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

#define MAXSIZE  32768L

int fh, result;
      .
      .
      .
fh = open("data",O_RDWR|O_CREAT,S_IREAD|S_IWRITE);
      .
      .
      .
/* Make sure the file is no longer than 32k before
** closing it.
*/

if (lseek(fh,0L,2) > MAXSIZE)
        result = chsize(fh,MAXSIZE);
```

## Summary

**#include <stdio.h>**

**void clearerr** (*stream*);
**FILE** **stream*; /* pointer to file structure */

## Description

The **clearerr** function resets the error indicator and end-of-file indicator for the specified *stream* to 0. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until **clearerr** or **rewind** is called.

## See Also

**eof, feof, ferror, perror**

## Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;
     .
     .
     .

/* The following statements output data to a
** stream and then check to make sure a write error has
** not occurred. The stream must previously have been
** opened for writing.
*/

if ((c=getc(stream)) == EOF) {
        if (ferror(stream)) {
                perror("write error");
                clearerr(stream);
                }
}
```

# close

## Summary

```
#include <io.h>          /* required only for function declarations */

int close(handle);
int handle;              /* handle referring to open file */
```

## Description

The **close** function closes the file associated with *handle*.

## Return Value

The **close** function returns 0 if the file was successfully closed. A return value of −1 indicates an error, and **errno** is set to **EBADF**, indicating an invalid file handle argument.

## See Also

**chsize, creat, dup, dup2, open, unlink**

## Example

```
#include <io.h>
#include <fcntl.h>

int fh;

fh = open("data",O_RDONLY);
      .
      .
close(fh);
```

## Summary

```
#include <math.h>

double cos(x);          /* calculate cosine of x */

double cosh(x);         /* calculate hyperbolic cosine of x */

double x;               /* radians */
```

## Description

The **cos** and **cosh** functions return the cosine and hyperbolic cosine of *x*, respectively.

## Return Value

**Cos** returns the cosine of *x*. If *x* is large, a partial loss of significance in the result may occur. In such cases, **cos** generates a **PLOSS** error, but no message is printed. If *x* is so large that a total loss of significance results, **cos** prints a **TLOSS** error message to **sdterr** and returns 0. In both cases, **errno** is set to **ERANGE**.

**Cosh** returns the hyperbolic cosine of *x*. If the result is too large, **cosh** returns the value **HUGE** and sets **errno** to **ERANGE**. Error handling can be modified by using the **matherr** routine.

## See Also

**acos, asin, atan, atan2, matherr, sin, sinh, tan, tanh**

## Example

```
#include <math.h>

double x, y;
     .
     .
     .
y = cos(x);
y = cosh(x);
```

# cprintf

## Summary

```
#include <conio.h>          /* required only for function declarations */

int cprintf(format-string[, argument... ]);
char *format-string;        /* format control string*/
```

## Description

The **cprintf** function formats and prints a series of characters and values directly to the console, using the **putch** function to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in the *format-string*. The *format-string* has the same form and function as the *format-string* argument for the **printf** function; see the **printf** reference page for a description of the *format-string* and arguments.

## Return Value

**Cprintf** returns the number of characters printed.

## See Also

**fprintf, printf, sprintf**

---

*Note*

Unlike the **fprintf**, **printf**, and **sprintf** functions, **cprintf** does not translate linefeed (LF) characters into carriage return/linefeed combinations (CR-LF) on output.

---

## Example

```
#include <conio.h>

int i = -16, j = 29;
unsigned int k = 511;

/* The following statement prints i=-16, j=0x1d, k=511 */

cprintf("i=%d, j=%#x, k=%u\n",i,j,k);
```

## cputs

### Summary

```
#include <conio.h>    /* required only for function declarations */

void cputs(str);
char *str;            /* pointer to output string */
```

### Description

The **cputs** function writes the null-terminated string pointed to by *str* directly to the console. Note that a carriage return/linefeed combination (CR-LF) is not automatically appended to the string after writing.

### Return Value

There is no return value.

### See Also

**putch**

### Example

```
#include <conio.h>

char *buffer = "Insert data disk in drive a: \r\n";

/* The following statement outputs a prompt to the
** console.
*/

cputs(buffer);
```

## Summary

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>                    /* required only for function declarations */

int creat(pathname, pmode);
char *pathname;                    /* pathname of new file */
int pmode;                         /* permission setting */
```

## Description

The **creat** function either creates a new file or opens and truncates an existing file. If the file specified by *pathname* does not exist, a new file is created with the given permission setting and opened for writing. If the file already exists and its permission setting allows writing, **creat** truncates the file to length 0, destroying the previous contents, and opens it for writing.

The permission setting, *pmode,* applies to newly created files only. The new file receives the specified permission setting after it is closed for the first time. *Pmode* is an integer expression containing one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in *sys\stat.h*. When both constants are given, they are joined with the bitwise OR operator (|). The meaning of the *pmode* argument is as follows.

| Value | Meaning |
|---|---|
| **S_IWRITE** | Writing permitted |
| **S_IREAD** | Reading permitted |
| **S_IREAD \| S_IWRITE** | Reading and writing permitted |

If write permission is not given, the file is read-only. Under MS-DOS it is not possible to give write-only permission. Thus, the modes **S_IWRITE** and **S_IREAD | S_IWRITE** are equivalent. Under MS-DOS Version 3.0 and later, files opened using **creat** are always opened in compatibility mode (see **sopen**).

**Creat** applies the current file permission mask to *pmode* before setting the permissions (see **umask**).

## Return Value

**Creat** returns a handle for the created file if the call is successful. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Pathname specifies an existing read-only file or specifies a directory instead of a file. |
| EMFILE | No more file handles available (too many open files). |
| ENOENT | Pathname not found. |

## See Also

**chmod, chsize, close, dup, dup2, open, sopen, umask**

---

*Note*

The **creat** routine is provided primarily for compatibility with previous libraries. A call to **open** with the **O_CREAT** and **O_TRUNC** values specified in the *oflag* argument is equivalent and is preferable for new code.

---

## Example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdlib.h>

int fh;

fh = creat("data",S_IREAD|S_IWRITE);

if (fh == -1)
        perror("couldn't create data file");
```

## Summary

```
#include <conio.h>          /* required only for function declarations */

int cscanf(format-string[, argument... ]);
char *format-string;        /* format control string */
```

## Description

The **cscanf** function reads data directly from the console into the locations given by the *arguments* (if any), using the **getche** function to read characters. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the **scanf** function; see the **scanf** reference page for a description of the *format-string*.

## Return Value

**Cscanf** returns the number of fields that were successfully converted and assigned. The return value does not include fields which were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

## See Also

**fscanf, scanf, sscanf**

## Example

```
#include <conio.h>

int result;
char buffer[20];
   .
   .
   .
cprintf("Please enter file name: ");

/* The following statement stores string input
** from the keyboard.
*/

result = cscanf("%19s",buffer);

/* Result is the number of correctly matched input
** fields. It is zero if none could be matched.
*/
```

## Summary

```
#include <time.h>          /* required only for function declarations */

char *ctime(time);
long *time;                /* pointer to stored time */
```

## Description

The **ctime** function converts a time stored as a **long** value to a character string. The *time* value is usually obtained from a call to *time,* which returns the number of seconds elapsed since 00:00:00 Greenwich Mean Time, January 1, 1970.

The string result produced by **ctime** contains exactly 26 characters and has the form of the following example.

```
Mon Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The newline character ('\n') and the null character ('\0') occupy the last two positions of the string.

Under MS-DOS, dates prior to 1980 are not understood. If *time* represents a date before January 1, 1980, **ctime** returns the character string representation of 00:00:00 January 1, 1980.

## Return Value

**Ctime** returns a pointer to the character string result. There is no error return.

## See Also

**asctime, ftime, gmtime, localtime, time**

---

*Note*

The **asctime** and **ctime** functions use a single statically allocated buffer for holding the return string. Each call to one of these routines destroys the result of the previous call.

---

## Example

```
#include <time.h>
#include <stdio.h>

long ltime;

time(&ltime);
printf("the time is %s\n",ctime(&ltime));
```

## Summary

**#include <dos.h>**

**int dosexterr** (*buffer*);
**struct DOSERROR** **buffer*;

## Description

The **dosexterr** function obtains the register values returned by the MS-DOS system call 59H and stores the values in the structure pointed to by *buffer*. This function is useful when making system calls under MS-DOS Version 3.0 or later, which offers extended error handling. See your *Microsoft MS-DOS Programmer's Reference Manual* for details on MS-DOS system calls.

The structure type **DOSERROR** is defined in *dos.h* as follows.

```
struct DOSERROR {
        int exterror;
        char class;
        char action;
        char locus;
        };
```

Giving a **NULL** pointer argument causes **dosexterr** to return the value in AX without filling in the structure fields.

## Return Value

The **dosexterr** function returns the value in the AX register (identical to the value in the **exterror** structure field).

## See Also

**perror**

---

*Note*

The **dosexterr** function should only be used under MS-DOS Version 3.0 or later.

---

## Example

```
#include <dos.h>
#include <fcntl.h>
#include <stdio.h>

struct DOSERROR doserror;
int fd;

if ((fd = open("test.dat", O_RDONLY)) == -1) {
        dosexterr(&doserror);
        printf("error=%d, class=%d, action=%d, locus=%d\n",
                doserror.exterror, doserror.class,
                doserror.action, doserror.locus);
        }
```

## Summary

```
#include <io.h>                    /* required only for function declarations */

int dup(handle);                   /* create second handle for open file */
int handle;                        /* handle referring to open file */
                                   /* force handle2 to refer to handle1 file */
int dup2(handle1, handle2);

int handle1;                       /* handle referring to open file */
int handle2;                       /* any handle value */
```

## Description

The **dup** and **dup2** functions cause a second file handle to be associated with a currently open file. Operations on the file can be carried out using either file handle, since all handles associated with a given file use the same file pointer. The type of access allowed for the file is unaffected by the creation of a new handle.

**Dup** returns the next available file handle for the given file. **Dup2** forces the given handle, *handle2,* to refer to the same file as *handle1.* If *handle2* is associated with an open file at the time of the call, that file is closed.

## Return Value

**Dup** returns a new file handle. **Dup2** returns 0 to indicate success. Both functions return −1 if an error occurs, and set **errno** to one of the following values.

| Value | Meaning |
|---|---|
| EBADF | Invalid file handle |
| EMFILE | No more file handles available (too many open files) |

## See Also

close, creat, open

## Example

```
#include <io.h>
#include <stdlib.h>

int fh;
     .
     .
     .

/* Get another file handle to refer to the same file as
** file handle 1 (stdout).
*/

fh = dup(1);

if (fh == -1)
        perror("dup(1) failure");

/* Now make file handle 3 refer to the same file as file
** handle 1 (stdout). If file handle 3 is already open,
** it is closed first.
*/

fh = dup2(1,3);

if (fh != 0)
        perror("dup2(1,3) failure");
```

## Summary

```
#include <stdlib.h>   /* required only for function declarations */

char *ecvt(value, ndigits, decptr, signptr);
double value;          /* number to be converted */
int ndigits;           /* number of digits stored */
int *decptr;           /* pointer to stored decimal point position */
int *signptr;          /* pointer to stored sign indicator */
```

## Description

The **ecvt** function converts a floating-point number to a character string. *Value* is the floating-point number to be converted. **Ecvt** stores *ndigits* digits of *value* as a string and appends a null character ('\0'). If the number of digits in *value* exceeds *ndigits,* the low-order digit is rounded. If there are fewer than *ndigits* digits, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* may be obtained after the call from *decptr* and *signptr*. *Decptr* points to an integer value giving the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. *Signptr* points to an integer indicating the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is is negative.

## Return Value

**Ecvt** returns a pointer to the string of digits. There is no error return.

## See Also

atof, atoi, atol, fcvt, gcvt

---

*Note*

The **ecvt** and **fcvt** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

---

## Example

```
#include <stdlib.h>

int decimal, sign;
char *buffer;
int precision = 10;

buffer = ecvt(3.1415926535, precision, &decimal, &sign);
    /* buffer contains "3141592654", decimal = 1, sign = 0 */
```

## Summary

```
#include <io.h>          /* required only for function declarations */

int eof(handle);
int handle;              /* handle referring to open file */
```

## Description

The **eof** function determines whether end-of-file has been reached for the file associated with *handle*.

## Return Value

**Eof** returns the value 1 if the current position is end-of-file, 0 if it is not. A return value of −1 indicates an error; in this case **errno** is set to **EBADF**, indicating an invalid file handle.

## See Also

**clearerr, feof, ferror, perror**

## eof

### Example

```
#include <io.h>
#include <fcntl.h>

int fh, count;
char buf[10];

fh = open("data",O_RDONLY);
          .
          .
          .

/* The following statement tests for an end-of-file condition
** before reading.
*/

while (!eof(fh)) {
          count = read(fh, buf, 10);
           .
           .
           .
          }
```

## Summary

```
#include <process.h>          /* required only for function declarations */
int execl(pathname, arg0, arg1...,argn, NULL);
int execle(pathname, arg0, arg1...,argn, NULL, envp);
int execlp(pathname, arg0, arg1...,argn, NULL);
int execv(pathname, argv);
int execve(pathname, argv, envp);
int execvp(pathname, argv);

char *pathname;              /* pathname of file to be executed */
char *arg0,*arg1,...,*argn;  /* list of pointers to arguments */
char *argv[ ];               /* array of pointers to arguments */
char *envp[ ];               /* array of pointers to environment settings */
```

## Description

The **exec** functions load and execute new child processes. When the call is successful, the child process is placed in the memory previously occupied by the calling process. Sufficient memory must be available for loading and executing the child process.

The *pathname* argument specifies the file to be executed as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a filename. If *pathname* does not have a filename extension or does not end with a period (.), the **exec** calls first append the extension ".COM" and search for the file; if unsuccessful, the extension ".EXE" is attempted. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the **exec** calls search for *pathname* with no extension. The **execlp** and **execvp** routines search for *pathname* (using the same procedures) in the directories specified by the PATH environment variable.

Arguments are passed to the new process by giving one or more pointers to character strings as arguments in the **exec** call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the new process must not exceed 128 bytes. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are counted.

The argument pointers may be passed as separate arguments (**execl, execle**, and **execlp**) or as an array of pointers (**execv, execve**, and **execvp**). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this argument is a copy of the *pathname* argument. (A different value will not produce an error). Under versions of MS-DOS earlier than 3.0, the passed value of *arg0* or *arg*[0] is not available for use in the child process. However, under MS-DOS 3.0 and later, the *pathname* is available as *arg0* or *arg*[0].

The **execl, execle** and **execlp** calls are typically used in cases where the number of arguments is known in advance. *Arg0* is usually a pointer to *pathname*. *Arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn* there must be a **NULL** pointer to mark the end of the argument list.

**Execv, execve,** and **execvp** are useful when the number of arguments to the new process is variable. Pointers to the arguments are passed as an array, *argv*. *Argv*[0] is usually a pointer to *pathname*. *Argv*[1] through *argv*[*n*] are pointers to the character strings forming the new argument list. *Argv*[*n*+1] must be a **NULL** pointer to mark the end of the argument list.

Files that are open when an **exec** call is made remain open in the new process. In the **execl, execlp, execv**, and **execvp** calls, the child process inherits the environment of the parent. **Execle** and **execve** allow the user to alter the environment for the child process by passing a list of environment settings through the *envp* argument. *Envp* is an array of character pointers, each element of which points to a null-terminated string defining an environment variable. Such a string usually has the following form

NAME=*value*

where NAME is the name of an environment variable and *value* is the string value to which that variable is set. (Notice that *value* is not enclosed in double quotes.) When *envp* is **NULL**, the child process inherits the environment settings of the parent process.

## Return Value

The **exec** functions do not normally return to the calling process. If an **exec** function returns, an error has occurred and the return value is −1. The **errno** variable is set to one of the following values.

| Value | Meaning |
|---|---|
| E2BIG | The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K bytes. |
| EACCES | Locking or sharing violation on the specified file (MS-DOS Version 3.0 or later). |
| EMFILE | Too many open files (the specified file must be opened to determine whether it is executable.) |
| ENOENT | File or pathname not found. |
| ENOEXEC | The specified file is not executable or has an invalid executable file format. |
| ENOMEM | Not enough memory is available to execute the child process; or the available memory has been corrupted; or an invalid block exists, indicating that the parent process was not allocated properly. |

## See Also

**abort, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp, system**

---

*Note*

The **exec** calls do not preserve the translation modes of open files. If the child process must use files inherited from the parent, the **setmode** routine should be used to set the translation mode of these files to the desired mode.

Signal settings are not preserved in child processes created by calls to **exec** routines. The signal settings are reset to the default in the child process.

---

## Example

```
#include <process.h>
#include <stdio.h>

extern char **environ;

char *args[4];
int result;

args[0] = "child";
args[1] = "one";
args[2] = "two";
args[3] = NULL;

/* All of the following statements attempt to execute a
** process called "child.exe" and pass it 3 arguments.
*/

result = execl("child.exe","child","one","two",NULL);
result = execle("child.exe","child","one","two",NULL,
                environ);
result = execlp("child.exe","child","one","two",NULL);
result = execv("child.exe",args);
result = execve("child.exe",args,environ);
result = execvp("child.exe",args);
```

## Summary

```
#include <process.h>        /* required only for function declarations */
void exit(status);          /* terminate after closing files */
void _exit(status);         /* terminate without flushing stream buffers */
int status;                 /* exit status */
```

## Description

The **exit** and **_exit** functions terminate the calling process. **Exit** flushes all buffers and closes all open files before terminating the process. **_Exit** terminates the process without flushing stream buffers. *Status* is typically given the value 0 to indicate a normal exit and set to some other value to indicate an error.

Although the **exit** and **_exit** calls do not return a value, the low-order byte of *status* is made available to the waiting parent process, if there is one, after the calling process exits. If there is no parent process waiting on the exiting process, the *status* value is lost.

## Return Value

There is no return value.

## See Also

**abort, execl, execle, execlp, execv, execve, execvp, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp**

## Example

```
#include <process.h>
#include <stdio.h>

FILE *stream;
        .
        .
        .
/* The following statements cause the process to
** terminate, after flushing buffers and closing
** open files, if another file cannot be opened.
*/

if ((stream = fopen("data","r")) == NULL) {
        perror("couldn't open data file");
        exit(1);
        }

/* The following statements cause the process to
** terminate immediately if a file cannot be opened.
*/

if ((stream = fopen("data","r")) == NULL) {
        perror("couldn't open data file");
        _exit(1);
        }
```

## Summary

```
#include <math.h>

double exp(x);
double x;              /* floating-point value */
```

## Description

The **exp** function returns the exponential function of its floating-point argument $x$.

## Return Value

**Exp** returns $\mathbf{e}^x$. On overflow, the function returns **HUGE**; on underflow, it returns 0. In both cases **errno** is set to **ERANGE**.

## See Also

log

## Example

```
#include <math.h>

double x, y;
     .
     .
     .
y = exp(x);
```

# fabs

## Summary

```
#include <math.h>

double fabs(x);
double x;                    /* floating-point value */
```

## Description

The **fabs** function returns the absolute value of its floating-point argument.

## Return Value

**Fabs** returns the absolute value of its argument. There is no error return.

## See Also

**abs, cabs, labs**

## Example

```
#include <math.h>

double x, y;
     .
     .
     .
y = fabs(x);
```

## Summary

```
#include <stdio.h>

int fclose(stream);                    /* close an open stream */
FILE *stream;                          /* pointer to file structure */

int fcloseall( );                      /* close all open streams */
```

## Description

The **fclose** and **fcloseall** functions close a stream or streams. All buffers associated with the stream(s) are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned using **setbuf** are not automatically released.

The **fclose** function closes the given *stream*. The **fcloseall** function closes all open streams except **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**.

## Return Value

**Fclose** returns 0 if the stream is successfully closed. **Fcloseall** returns the total number of streams closed. Both functions return **EOF** to indicate an error.

## See Also

**close, fdopen, fflush, fopen, freopen**

## Example

```
#include <stdio.h>

FILE *stream;
int numclosed;

stream = fopen("data","r");
    .
    .
    .

/* The following statement closes the stream.
*/

fclose(stream);


/* The following statement closes all streams except
** stdin, stdout, stderr, stdaux, and stdprn.
*/

numclosed = fcloseall();
```

## Summary

```
#include <stdlib.h>          /* required only for function declarations */

char *fcvt(value, ndec, decptr, signptr);
double value;                /* number to be converted */
int ndec;                    /* number of digits after decimal point */
int *decptr;                 /* pointer to stored decimal point position */
int *signptr;                /* pointer to stored sign indicator */
```

## Description

The **fcvt** function converts a floating-point number to a character string. *Value* is the floating-point number to be converted. **Fcvt** stores the digits of *value* as a string and appends a null character ('\0'). *Ndec* specifies the number of digits to be stored after the decimal point.

If the number of digits after the decimal point in *value* exceeds *ndec,* the correct digit is rounded according to the FORTRAN F format. If there are fewer than *ndec* digits of precision, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* may be obtained after the call from *decptr* and *signptr*. *Decptr* points to an integer value giving the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. *Signptr* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive, and is set to a nonzero number if *value* is negative.

## Return Value

**Fcvt** returns a pointer to the string of digits. There is no error return.

## See Also

**atof, atoi, atol, ecvt, gcvt**

---

*Note*

The **ecvt** and **fcvt** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

---

## Example

```
#include <stdlib.h>

int decimal, sign;
char *buffer;
int precision = 10;

buffer = fcvt(3.1415926535,precision,&decimal,&sign);

/* buffer = "31415926535", decimal = 1, sign = 0 */
```

## Summary

```
#include <stdio.h>

FILE *fdopen(handle, type);
int handle;                    /* handle referring to open file */
char *type;                    /* type of access permitted */
```

## Description

The **fdopen** function associates an input/output stream with the file identified by *handle*. *Type* is a character string specifying the type of access requested for the file, as follows:

| Type | Description |
|---|---|
| "r" | Open for reading (the file must exist). |
| "w" | Open an empty file for writing; if the given file exists, its contents are destroyed. |
| "a" | Open for writing at the end of the file (appending); create the file first if it doesn't exist. |
| "r+" | Open for both reading and writing (the file must exist). |
| "w+" | Open an empty file for both reading and writing; if the given file exists, its contents are destroyed. |
| "a+" | Open for reading and appending; create the file first if it doesn't exist. |

---

*Note*

Use the "w" and "w+" modes with care as they can destroy existing files.

---

The specified *type* must be compatible with the access mode with which the file was opened.

When a file is opened with "a" or "a+" type, all write operations take place at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching from reading to writing or vice versa, there must be an intervening **fseek** or **rewind** operation. The current position can be specified for the **fseek** operation if desired.

## Return Value

**Fdopen** returns a pointer to the open stream. A **NULL** pointer value indicates an error.

## See Also

**dup, dup2, fclose, fcloseall, fopen, freopen, open**

## Example

```
#include <stdio.h>
#include <fcntl.h>

FILE *stream;
int fh;

fh = open("data",O_RDONLY);

/* The following statement associates a stream with the
** open file handle.
*/

stream = fdopen(fh,"r");
```

## Summary

```
#include <stdio.h>

int feof(stream);
FILE *stream;          /* pointer to file structure */
```

## Description

The **feof** function determines whether the end of the given *stream* has been reached. Once end-of-file is reached, read operations return an end-of-file indicator until the stream is closed or **rewind** is called.

## Return Value

**Feof** returns a nonzero value when the current position is end-of-file. The value 0 is returned if the current position is not end-of-file. There is no error return.

## See Also

**clearerr, eof, ferror, perror**

---

*Note*

**Feof** is implemented as a macro.

---

### Example

```
#include <stdio.h>

char string[100];
FILE *stream;
   .
   .
   .

/* The following statements process lines of input
** until eof occurs.
*/

while (!feof(stream)) {
          if (fscanf(stream," %s ",string))
                    process(string);
          }
```

## Summary

```
#include <stdio.h>

int ferror(stream);
FILE *stream;       /* pointer to file structure */
```

## Description

The **ferror** function tests for a reading or writing error on the given *stream*. If an error has occurred, the error indicator for the *stream* remains set until the stream is closed or rewound or until **clearerr** is called.

## Return Value

**Ferror** returns a nonzero value to indicate an error on the given *stream*. The return value 0 means no error has occurred.

## See Also

**clearerr, eof, feof, fopen, perror**

---

*Note*

**Ferror** is implemented as a macro.

---

## Example

```
#include <stdio.h>

FILE *stream;
char *string;
   .
   .
   .

/* The following statements output data to a
** stream and then check to make sure a write error has
** not occurred. The stream must previously have been
** opened for writing.
*/

fprintf(stream,"%s\n",string);
if (ferror(stream)) {
        perror("write error");
        clearerr(stream);
        }
```

## Summary

```
#include <stdio.h>

int fflush(stream);
FILE *stream;       /* pointer to file structure */
```

## Description

The **fflush** function causes the contents of the buffer associated with the specified output *stream* to be written to the associated file. The *stream* remains open after the call. **Fflush** has no effect on an unbuffered stream.

## Return Value

**Fflush** returns the value 0 if the buffer was successfully flushed. The value 0 is also returned in cases where the specified stream has no buffer or is open for reading only. A return value of **EOF** indicates an error.

## See Also

**fclose, flushall, setbuf**

---

*Note*

Buffers are automatically flushed when they are full, when the stream is closed, or when a program terminates normally without closing the stream.

---

## Example

```
#include <stdio.h>

FILE *stream;
char buffer[BUFSIZ];
   .
   .
   .

/* The following statements flush a stream's buffer and
** set up a new buffer for that stream.
*/

fflush(stream);
setbuf(stream,buffer);
```

## Summary

```
#include <stdio.h>
int fgetc(stream);      /* read a character from stream */
FILE *stream;           /* pointer to file structure */

int fgetchar( );        /* read a character from stdin */
```

## Description

The **fgetc** function reads a single character from the input *stream* at the current position and increments the associated file pointer (if any) to point to the next character. **Fgetchar** is equivalent to **fgetc(stdin)**.

## Return Value

**Fgetc** and **fgetchar** return the character read. A return value of **EOF** may indicate an error or end-of-file; however, the **EOF** value is also a legitimate integer value, so **feof** or **ferror** should be used to verify an error or end-of-file condition.

## See Also

**fputc, fputchar, getc, getchar**

---

*Note*

**Fgetc** and **fgetchar** are identical to **getc** and **getchar**, but are functions, not macros.

---

## Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;
   .
   .
   .
/* The following statements gather a line of input from
** a stream.
*/

for (i = 0; (i < 80) && ((ch = fgetc(stream)) != EOF) &&
        (ch != '\n'); i++)
        buffer[i] = ch;

buffer[i] = '\0';

/* "fgetchar()" could be used instead of "fgetc(stream)" in
** the for statement above to gather a line of input from
** stdin (equivalent to "fgetc(stdin)".
*/
```

## Summary

```
#include <stdio.h>

char *fgets(string, n, stream);          /* read a string from stream */
char *string;                            /* storage location for data */
int n;                                   /* number of characters stored */
FILE *stream;                            /* pointer to file structure */
```

## Description

The **fgets** function reads a string from the input *stream* and stores it in *string*. Characters are read from the current *stream* position up to and including the first newline character ('\n'), up to the end of the stream, or until the number of characters read is equal to $n-1$, whichever comes first. The result is stored in *string*, and a null character ('\0') is appended. The newline, if read, is included in the *string*. If $n$ is equal to 1, *string* is empty ("").

## Return Value

**Fgets** returns *string*. A **NULL** return value indicates an error or end-of-file condition. Use **feof** or **ferror** to determine whether the **NULL** value represents an error or end-of-file.

## See Also

**fputs, gets, puts**

## Example

```
#include <stdio.h>

FILE *stream;
char line[100], *result;
   .
   .
   .

/* The following statement gets a line of input from a
** stream. No more than 99 characters, or up to \n,
** are read from the stream.
*/

result = fgets(line,100,stream);
```

## Summary

```
#include <io.h>                 /* required only for function declarations */

long filelength(handle);
int handle;                     /* handle referring to open file */
```

## Description

The **filelength** function returns the length in bytes of the file associated with the given *handle*.

## Return Value

**Filelength** returns the file length in bytes. A return value of −1L indicates an error, and **errno** is set to **EBADF** to indicate an invalid file handle.

## See Also

**chsize, fileno, fstat, stat**

## Example

```
#include <io.h>
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
long length;

stream = fopen("data","r");
   .
   .
   .
/* The following statements attempt to determine the
** length of a file associated with a stream.
*/

length = filelength(fileno(stream));

if (length == -1L)
        perror("filelength failed");
```

# fileno

## Summary

```
#include <stdio.h>

int fileno(stream);
FILE *stream;       /* pointer to file structure */
```

## Description

The **fileno** function returns the file handle currently associated with the given *stream*. If more than one handle is associated with the stream, the return value is the handle assigned when the stream was first opened.

## Return Value

**Fileno** returns the file handle. There is no error return. The result is undefined if *stream* does not specify an open file.

## See Also

**fdopen, filelength, fopen, freopen**

---

*Note*

**Fileno** is implemented as a macro.

---

## Example

```
#include <stdio.h>

int result;

/* The following statement determines the file handle
** of the stderr stream.
*/

result = fileno(stderr);    /* result is 2 */
```

# floor

## Summary

```
#include <math.h>

double floor(x);
double x;              /* floating-point value */
```

## Description

The **floor** function returns a floating-point value representing the largest integer that is less than or equal to *x*.

## Return Value

**Floor** returns the floating-point result. There is no error return.

## See Also

**ceil**, **fmod**

## Example

```
#include <math.h>

double y;
     .
     .
     .
y = floor(2.8);          /* y = 2.0 */
y = floor(-2.8);         /* y = -3.0 */
```

## Summary

```
#include <stdio.h>

int flushall( );
```

## Description

The function **flushall** causes the contents of all buffers associated with open output streams to be written to the associated files. All streams remain open after the call.

## Return Value

**Flushall** returns the number of open streams (input and output). There is no error return.

## See Also

**fflush**

---

*Note*

Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.

---

## Example

```
#include <stdio.h>

int numflushed;
   .
   .
   .

/* The following statement resolves any pending i/o on
** all streams.
*/

numflushed = flushall();
```

## Summary

```
#include <math.h>

double fmod(x,y);
double x;                    /* floating-point values */
double y;
```

## Description

The **fmod** function calculates the floating-point remainder of $x/y$, such that $x = iy + f$, where $i$ is an integer, $f$ has the same sign as $x$, and the absolute value of $x$ is less than the absolute value of $y$.

## Return Value

**Fmod** returns the floating-point remainder. If $y$ is zero or if $x/y$ causes an overflow, the function returns 0.

## See Also

**ceil, fabs, floor**

## Example

```
#include <math.h>

double x, y, z;

x = -10.0;
y = 3.0;
z = fmod(x,y);                    /* z = -1.0 */
```

# fopen

## Summary

```
#include <stdio.h>

FILE *fopen(pathname, type);
char *pathname;                    /* pathname of file */
char *type;                        /* type of access permitted */
```

## Description

The **fopen** function opens the file specified by *pathname*. *Type* is a character string specifying the type of access requested for the file, as follows:

| Type | Description |
|---|---|
| "r" | Open for reading (the file must exist). |
| "w" | Open an empty file for writing; if the given file exists, its contents are destroyed. |
| "a" | Open for writing at the end of the file (appending); create the file first if it doesn't exist. |
| "r+" | Open for both reading and writing (the file must exist). |
| "w+" | Open an empty file for both reading and writing; if the given file exists, its contents are destroyed. |
| "a+" | Open for reading and appending; create the file first if it doesn't exist. |

---

*Note*

Use the "w" and "w+" modes with care as they can destroy existing files.

---

When a file is opened with "a" or "a+" type, all write operations take place at the end of the file. Although the file pointer can be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching between reading and writing, there must be an intervening **fseek** or **rewind** operation. The current position may be specified for the **fseek** operation if desired.

In addition to the values listed above, one of the following characters may be appended to the *type* string to specify the translation mode for newlines.

| Character | Meaning |
|---|---|
| t | Open in text (translated) mode; carriage return/linefeed combinations (CR-LF) are translated into a single linefeed (LF) on input; linefeed characters are translated to carriage return/linefeed combinations on output. |
| b | Open in binary (untranslated) mode; the above translations are suppressed. |

If "t" or "b" is not given in the *type* string, the translation mode is defined by the default mode variable **_fmode**.

## Return Value

**Fopen** returns a pointer to the open file. A **NULL** pointer value indicates an error.

## See Also

**fclose, fcloseall, fdopen, ferror, fileno, freopen, open, setmode**

## Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

/* The following statements attempt to open a file for
** reading.
*/

if ((stream = fopen("data","r")) == NULL)
        perror("couldn't open data file");
```

## Summary

**#include <dos.h>**

**unsigned FP_OFF**(*longptr*);

**unsigned FP_SEG**(*longptr*);

**char far** **longptr*; /* long pointer to memory address */

## Description

The **FP_OFF** and **FP_SEG** macros return the offset and segment, respectively, of the long pointer *longptr*.

## Return Value

**FP_OFF** returns an unsigned integer value representing an offset.
**FP_SEG** returns an unsigned integer value representing a segment address.

## See Also

**segread**

## Example

```
#include <dos.h>

char far *p;
unsigned int sp;
unsigned int op;
      .
      .
      .
sp = FP_SEG;
op = FP_OFF;
```

# fprintf

## Summary

```
#include <stdio.h>

int fprintf(stream, format-string[, argument... ]);
FILE *stream;                                   /* pointer to file structure */
char *format-string;                            /* format control string*/
```

## Description

The **fprintf** function formats and prints a series of characters and values to the output *stream*. Each *argument* (if any) is converted and output according to the corresponding format specification in the *format-string*.

The *format-string* has the same form and function as the *format-string* argument for the **printf** function; see the **printf** reference page for a description of the *format-string* and *arguments*.

## Return Value

**Fprintf** returns the number of characters printed.

## See Also

**cprintf, fscanf, printf, sprintf**

## Example

```
#include <stdio.h>

FILE *stream;
int i = 10;
double fp = 1.5;
char *s = "this is a string";
char c = '\n';

stream = fopen("results","w");

/* Format and print various data. */

fprintf(stream, "%s%c",s,c);     /* prints "this is a string"
                                 ** followed by a newline
                                 */
fprintf(stream, "%d\n",i);      /* prints 10 followed by
                                 ** a newline
                                 */
fprintf(stream, "%f",fp);        /* prints 1.500000 */
```

# fputc – fputchar

## Summary

```
#include <stdio.h>

int fputc(c, stream);          /* write a character to  stream */
int c;                         /* character to be written */
FILE *stream;                  /* pointer to file structure */

int fputchar(c);               /* write a character to  stdout */
int c;                         /* character to be written */
```

## Description

The **fputc** function writes the single character *c* to the output *stream* at the current position. **Fputchar** is equivalent to **fputc**(*c*, **stdout**).

## Return Value

**Fputc** and **fputchar** return the character written. A return value of **EOF** may indicate an error; however, since the **EOF** value is also a legitimate integer value, use **ferror** to verify an error condition.

---

*Note*

**Fputc** and **fputchar** are identical to **putc** and **putchar**, but are functions, not macros.

---

## See Also

fgetc, fgetchar, putc, putchar

## Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i;
int ch;
   .
   .
   .

/* The following statements write the contents of a buffer to
** a stream.  Note that the output occurs as a side effect
** within the for statement's second expression, so the
** statement body is null.
*/

for (i = 0; (i < 81) &&
          ((ch = fputc(buffer[i],stream)) != EOF); i++)
          ;

/* "fputchar()" could be used instead of "fputc(stream)"
** in the for statement above to write the buffer to stdout
** (equivalent to "fputc(stdout)".
*/
```

# fputs

## Summary

```
#include <stdio.h>

int fputs(string, stream);      /* write a string to stream */
char *string;                   /* string to be output */
FILE *stream;                   /* pointer to file structure */
```

## Description

The **fputs** function copies *string* to the output *stream* at the current position. The terminating null character ('\0') is not copied.

## Return Value

**Fputs** returns the last character output. If the input *string* is empty, the return value is 0. The return value **EOF** indicates an error.

## See Also

**fgets, gets, puts**

## Example

```
#include <stdio.h>

FILE *stream;
int result;
   .
   .
   .

/* The following statement writes a string to a stream.
*/

result = fputs("data files have been updated\n",stream);
```

## Summary

```
#include <stdio.h>

int fread(buffer, size, count, stream);
char *buffer;                    /* storage location for data */
int size;                        /* item size in bytes */
int count;                       /* maximum number of items to be read */
FILE *stream;                    /* pointer to file structure */
```

## Description

The **fread** function reads up to *count* items of length *size* from the input *stream* and stores them in the given *buffer*. The file pointer associated with *stream* (if there is one) is incremented by the number of bytes actually read.

If the given *stream* was opened in text mode, carriage return/linefeed pairs (CR-LF) are replaced with single linefeed characters (LF). The replacement has no effect on the file pointer or the return value.

## Return Value

**Fread** returns the number of full items actually read, which may be less than *count* if an error occurs or the file end is encountered before reaching *count*.

## See Also

**fwrite, read**

## Example

```
#include <stdio.h>

FILE *stream;
long list[100];
int numread;

stream = fopen("data", "r+b");
   .
   .
   .

/* The following statement reads 100 binary long integers
** from the stream.
*/

numread = fread((char *)list,sizeof(long),100,stream);
```

## Summary

```
#include <malloc.h>          /* required only for function declarations */

void free(ptr);
char *ptr;                   /* pointer to allocated memory block */
```

## Description

The **free** function de-allocates a memory block. *Ptr* points to a memory block previously allocated through a call to **calloc**, **malloc**, or **realloc**. The number of bytes freed is the number of bytes specified when the block was allocated (or re-allocated, in the case of **realloc**). After the call, the freed block is available for allocation.

## Return Value

There is no return value.

## See Also

**calloc**, **malloc**, **realloc**

---

*Note*

Attempting to free an invalid *ptr* (a pointer not allocated via **calloc**, **malloc**, or **realloc**) may affect subsequent allocation and cause errors.

---

## Example

```
#include <malloc.h>
#include <stdio.h>

char *alloc;

/* Allocate 100 bytes and then free them.
*/

alloc = malloc(100);
     .
     .
     .
if (alloc != NULL)          /* test for valid pointer */
        free(alloc);        /* free memory for the heap */
```

## Summary

```
#include <stdio.h>

FILE *freopen(pathname, type, stream);
char *pathname;                    /* pathname of new file */
char *type;                        /* type of access permitted */
FILE *stream;                      /* pointer to file structure */
```

## Description

The **freopen** function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *pathname*. **Freopen** is typically used to redirect the pre-opened files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn** to files specified by the user. The new file associated with *stream* is opened with the given *type,* which is a character string specifying the type of access requested for the file, as follows.

| Type | Description |
|---|---|
| "r" | Open for reading (the file must exist). |
| "w" | Open an empty file for writing; if the given file exists, its contents are destroyed. |
| "a" | Open for writing at the end of the file (appending); create the file first if it doesn't exist. |
| "r+" | Open for both reading and writing (the file must exist). |
| "w+" | Open an empty file for both reading and writing; if the given file exists, its contents are destroyed. |
| "a+" | Open for reading and appending; create the file first if it doesn't exist. |

---

*Note*

Use the "w" and "w+" modes with care as they can destroy existing files.

---

When a file is opened with "a" or "a+" type, all write operations take place at the end of the file. Although the file pointer may be repositioned using **fseek** or **rewind**, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching between reading and writing, there must be an intervening **fseek** or **rewind** operation. The current position may be specified for the **fseek** operation if desired.

In addition to the values listed above, one of the following characters may be appended to the *type* string to specify the translation mode for newlines.

| Character | Meaning |
|---|---|
| t | Open in text (translated) mode; carriage return/linefeed combinations (CR-LF) are translated into a single linefeed (LF) on input; linefeed characters are translated to carriage return/linefeed combinations on output. |
| b | Open in binary (untranslated) mode; the above translations are suppressed. |

If "t" or "b" is not given in the *type* string, the translation mode is defined by the default mode variable **_fmode**.

## Return Value

**Freopen** returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns a **NULL** pointer value.

## See Also

**fclose, fcloseall, fdopen, fileno, fopen, open, setmode**

## Example

```
#include <stdio.h>

FILE *stream;
   .
   .
   .

/* The following statement closes the stdout stream and
** reassigns its stream pointer.
*/

stream = freopen("data2","w+",stdout);
```

# frexp

## Summary

```
#include <math.h>

double frexp(x, expptr);
double x;                    /* floating-point value */
int *expptr;                 /* pointer to stored integer exponent */
```

## Description

The **frexp** function breaks down the floating-point value $x$ into a mantissa $m$ and an exponent $n$ such that the absolute value of $m$ is greater than or equal to 0.5 and less than 1.0 and $x = m*2^n$. The integer exponent $n$ is stored at the location pointed to by *expptr*.

## Return Value

**Frexp** returns the mantissa $m$. If $x$ is zero, the function returns 0 for both the mantissa and exponent. There is no error return.

## See Also

**ldexp, modf**

## Example

```
#include <math.h>

double x, y;
int n;
     .
     .
     .
x = 16.4;
          /* y will be .5125, n will be 5 */
y = frexp(x,&n);
```

## Summary

```
#include <stdio.h>

int fscanf(stream, format-string [, argument... ]);
FILE *stream;                          /* pointer to file structure */
char *format-string;                   /* format control string */
```

## Description

The **fscanf** function reads data from the current position of the specified *stream* into the locations given by *arguments* (if any). Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the **scanf** function; see the **scanf** reference page for a description of the *format-string*.

## Return Value

**Fscanf** returns the number of fields that were successfully converted and assigned. The return value does not include fields which were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

## See Also

**cscanf, fprintf, scanf, sscanf**

### Example

```
#include <stdio.h>

FILE *stream;
long l;
float fp;
char s[81];
char c;

stream = fopen("data","r");
   .
   .
   .
/* Input various data. */

fscanf(stream, "%s",s);
fscanf(stream, " %c",&c);
fscanf(stream, "%ld",&l);
fscanf(stream, "%f",&fp);
```

## Summary

```
#include <stdio.h>

int fseek(stream, offset, origin);
FILE *stream;                    /* pointer to file structure */
long offset;                     /* number of bytes from origin */
int origin;                      /* initial position */
```

## Description

The **fseek** function moves the file pointer (if any) associated with *stream* to a new location that is *offset* bytes from the *origin*. The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a read or a write.

*Origin* must be one of the following constants.

| Origin | Definition |
|---|---|
| 0 | Beginning of file |
| 1 | Current position of file pointer |
| 2 | End of file |

**Fseek** can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. However, an attempt to position the pointer before the beginning of the file causes an error.

## Return Value

**Fseek** returns the value 0 if the pointer was successfully moved. A non-zero return value indicates an error. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

## See Also

**ftell, lseek, rewind**

---

*Note*

For streams opened in text mode, **fseek** has limited use because carriage return-linefeed translations can cause **fseek** to produce unexpected results. The only **fseek** operations guaranteed to work on streams opened in text mode are seeking with an offset of zero relative to any of the origin values, or seeking from the beginning of the file with an offset value returned from a call to **ftell**.

---

## Example

```
#include <stdio.h>

FILE *stream;
int result;

stream = fopen("data","r");
   .
   .
   .

/* The following statement returns the file pointer to the
** beginning of the file.
*/

result = fseek(stream,0L,0);
```

## Summary

```
#include <sys\types.h>
#include <sys\stat.h>

int fstat(handle, buffer);
int handle;                 /* handle referring to open file */
struct stat *buffer;        /* pointer to structure to store results */
```

## Description

The **fstat** function obtains information about the open file associated with the given *handle* and stores it in the structure pointed to by *buffer*. The structure, whose type **stat** is defined in *sys\stat.h*, contains the following fields.

| Field | Value |
|---|---|
| **st_mode** | Bit mask for file mode information. **S_IFCHR** bit set if *handle* refers to a device. **S_IFREG** bit set if *handle* refers to an ordinary file. User read/write bits set according to the file's permission mode. |
| **st_dev** | Either drive number of the disk containing the file, or *handle* in the case of a device. |
| **st_rdev** | Either drive number of the disk containing the file, or *handle* in the case of a device (same as **st_dev**). |
| **st_nlink** | Always 1. |
| **st_size** | Size of the file in bytes. |
| **st_atime** | Time of last modification of file. |
| **st_mtime** | Time of last modification of file (same as **st_atime**). |
| **st_ctime** | Time of last modification of file (same as **st_atime** and **st_mtime**). |

There are three additional fields in the **stat** structure type that do not contain meaningful values under MS-DOS.

## Return Value

**Fstat** returns the value 0 if the file status information is obtained. A return value of −1 indicates an error; in this case, **errno** is set to **EBADF**, indicating an invalid file handle.

## See Also

**access, chmod, filelength, stat**

---

*Note*

If the given *handle* refers to a device, the size and time fields in the **stat** structure are not meaningful.

---

## Example

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

struct stat buf;
int fh, result;

fh = open("tmp/data",O_RDONLY);
      .
      .
      .
result = fstat(fh,&buf);

if (result == 0)
        printf("file size is %ld\n",buf.st_size);
```

## Summary

```
#include <stdio.h>

long ftell(stream);
FILE *stream;              /* pointer to file structure */
```

## Description

The **ftell** function gets the current position of the file pointer (if any) associated with *stream*. The position is expressed as an offset relative to the beginning of the *stream*.

## Return Value

**Ftell** returns the current position. A return value of −1L indicates an error. On devices incapable of seeking (such as terminals and printers), or when *stream* does not refer to an open file, the return value is undefined.

## See Also

**fseek, lseek, tell**

---

*Note*

The value returned by **ftell** may not reflect the physical byte offset for streams opened in text mode, since text mode causes carriage return-linefeed translation. Use **ftell** in conjunction with the **fseek** function to remember and return to file locations correctly.

---

## Example

```
#include <stdio.h>

FILE *stream;
long position;

stream = fopen("data","rb");
   .
   .
   .
position = ftell(stream);
```

## Summary

```
#include <sys\types.h>
#include <sys\timeb.h>

void ftime(timeptr);
struct timeb *timeptr;      /* pointer to structure defined in sys\timeb.h */
```

## Description

The **ftime** function gets the current time and stores it in the structure pointed to by *timeptr*. The **timeb** structure is defined in *sys\timeb.h*. It contains four fields: **time**, **millitm**, **timezone** and **dstflag**.

**Time** gives the time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. **Millitm** gives up to 1000 milliseconds of more-precise interval. **Timezone** gives the difference in minutes, moving westward, between Greenwich Mean Time and local time. The value of *timezone* is set from the value of the global variable **timezone** (see **tzset**). **Dstflag** is nonzero if Daylight Savings Time is currently in effect for the local time zone, as determined from the value of the global variable **daylight** (see **tzset**).

## Return Value

**Ftime** gives values to the fields in the structure pointed to by *timeptr*. It does not return a value.

## See Also

**asctime, ctime, gmtime, localtime, time, tzset**

## Example

```
#include <sys/types.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <time.h>

struct timeb timebuffer;

ftime(&timebuffer);
printf("the time is %s\n",ctime(&(timebuffer.time)));
```

## Summary

```
#include <stdio.h>

int fwrite(buffer, size, count, stream);
char *buffer;                    /* pointer to data to be written */
int size;                        /* item size in bytes */
int count;                       /* maximum number of items to be written */
FILE *stream;                    /* pointer to file structure */
```

## Description

The **fwrite** function writes up to *count* items of length *size* from *buffer* to the output *stream*. The file pointer associated with *stream* (if there is one) is incremented by the number of bytes actually written.

If the given *stream* was opened in text mode, each carriage return is replaced with a carriage return/linefeed pair. The replacement has no effect on the return value.

## Return Value

**Fwrite** returns the number of full items actually written, which may be less than *count* if an error occurs.

## See Also

**fread, write**

## Example

```
#include <stdio.h>

FILE *stream;
long list[100];
int numwritten;

stream = fopen("data", "r+b");
   .
   .
   .

/* The following statement writes 100 long integers to
** a stream in binary format.
*/

numwritten = fwrite((char *)list,sizeof(long),100,stream);
```

## Summary

```
#include <stdlib.h>          /* required only for function declarations */

char *gcvt(value, ndec, buffer);
double value;                /* value to be converted */
int ndec;                    /* number of significant digits stored */
char *buffer;                /* storage location for result */
```

## Description

The function **gcvt** converts a floating-point *value* to a character string and stores the string in *buffer*. The *buffer* should be large enough to accommodate the converted value plus a terminating null character ('\0'), which is automatically appended. There is no provision for overflow.

**Gcvt** attempts to produce *ndec* significant digits in FORTRAN F format. Failing that, it produces *ndec* significant digits in FORTRAN E format. Trailing zeros may be suppressed in the conversion.

## Return Value

**Gcvt** returns a pointer to the string of digits. There is no error return.

## See Also

**atof, atoi, atol, ecvt, fcvt**

## Example

```
#include <stdlib.h>

char buffer[50];
int precision = 7;
        /* buffer contains "-314150.0" */
gcvt(-3.1415e5,precision,buffer);
```

## getc – getchar

### Summary

```
#include <stdio.h>

int getc(stream);           /* read a character from stream */
FILE *stream;               /* pointer to file structure */

int getchar( );             /* read a character from stdin */
```

### Description

**Getc** reads a single character from the current *stream* position and increments the associated file pointer (if there is one) to point to the next character. **Getchar** is identical to **getc(stdin)**.

### Return Value

**Getc** and **getchar** return the character read. A return value of **EOF** indicates an error or end-of-file condition. Use **ferror** or **feof** to determine whether an error or end-of-file occurred.

### See Also

**fgetc, fgetchar, getch, getche, putc, putchar, ungetc**

---

*Note*

**Getc** and **getchar** are identical to **fgetc** and **fgetchar**, but are macros, not functions.

---

## Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i, ch;
          .
          .
          .

/* The following statements gather a line of input from
** stdin.
*/

for (i = 0; (i < 80) && ((ch = getchar()) != EOF) &&
          (ch != '\n'); i++)
          buffer[i] = ch;

buffer[i] = '\0';

/* "getc(stdin)" could be used instead of "getchar()" in the
** for statement above to gather a line of input from stdin
*/
```

# getch

## Summary

```
#include <conio.h>   /* required only for function declarations */

int getch( );
```

## Description

The **getch** function reads, without echoing, a single character directly from the console. Characters typed are not echoed. If a CONTROL-C is typed, the system executes an INT 23H (CONTROL-C exit).

## Return Value

**Getch** returns the character read. There is no error return.

## See Also

**cgets, getche, getchar**

## Example

```
#include <conio.h>
#include <ctype.h>

int ch;

/* This loop gets characters from the keyboard until a
** non-blank character is seen. Preceding blank
** characters are discarded.
*/

do      {
        ch = getch();
        } while (isspace(ch));
```

## Summary

```
#include <conio.h>   /* required only for function declarations */

int getche( );
```

## Description

The **getche** function reads a single character from the console and echoes the character read. If a CONTROL-C is typed, the system executes an INT 23H (CONTROL-C exit).

## Return Value

**Getche** returns the character read. There is no error return.

## See Also

**cgets, getch, getchar**

## Example

```
#include <conio.h>
#include <ctype.h>

int ch;

/* Get a character from the keyboard and echo it to the
** console. If it is an upper case letter, convert it
** to lower case and write over the old character.
*/

ch = getche();

if (isupper(ch))
        cprintf("\b%c",_tolower(ch));
```

# getcwd

## Summary

```
#include <direct.h>          /* required only for function declarations */

char *getcwd(pathbuf, n);
char *pathbuf;               /* storage location for pathname */
int n;                       /* maximum length of pathname */
```

## Description

The **getcwd** function gets the full pathname of the current working directory and stores it at *pathbuf*. The integer argument *n* specifies the maximum length for the pathname. An error occurs if the length of the pathname (including the terminating null character) exceeds *n*.

The *pathbuf* argument can be **NULL**; a buffer of size *n* will automatically be allocated (via **malloc**) and used to store the pathname. This buffer can later be freed by using the **getcwd** return value (a pointer to the allocated buffer) with the **free** function.

## Return Value

**Getcwd** returns *pathbuf*. A **NULL** return value indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| **ENOMEM** | Insufficient memory to allocate *n* bytes (when **NULL** argument given as *pathbuf*). |
| **ERANGE** | Pathname longer than *n* characters. |

## See Also

chdir, mkdir, rmdir

## Example

```
#include <direct.h>
#include <stdlib.h>

char buffer[51];

/* The following statement stores the name of the current
** working directory (up to 50 characters long) in buffer.
*/

if (getcwd(buffer,50) == NULL)
        perror("getcwd error");
```

# getenv

## Summary

```
#include <stdlib.h>          /* required only for function declarations */

char *getenv(varname);
char *varname;               /* name of environment variable */
```

## Description

The **getenv** function searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program).

## Return Value

**Getenv** returns a pointer to the environment table entry containing the current string value of *varname*. The return value is **NULL** if the given variable is not currently defined.

## See Also

**putenv**

---

*Note*

Environment table entries must not be changed directly. If an entry must be changed, use the **putenv** function. To modify the returned value without affecting the environment table, use **strdup** or **strcpy** to make a copy of the string.

The **getenv** and **putenv** functions use the global variable **environ** to access the environment table. **Putenv** may change the value of **environ**, thus invalidating the "envp" argument to the "main" function.

---

## Example

```
#include <stdlib.h>

char *pathvar;

/* The following statement gets the value of the PATH
** environment variable.
*/

pathvar = getenv("PATH");

/* If an entry such as "PATH=A:\BIN;B:\BIN" is in the
** environment, pathvar will point to "A:\BIN;B:\BIN". If
** there is no PATH environment variable, pathvar will
** be NULL.
*/
```

## getpid

### Summary

```
#include <process.h>        /* required only for function declarations */

int getpid( );
```

### Description

The **getpid** function returns an integer, the process ID, that uniquely identifies the calling process.

### Return Value

**Getpid** returns the process ID. There is no error return.

### See Also

**mktemp**

### Example

```
#include <process.h>
#include <string.h>
#include <stdio.h>

char filename[9], pid[5];
   .
   .
   .
strcpy(filename, "FILE");
strcat(filename, itoa(getpid(),pid,10));
      /* prints "FILExxxxx", where xxxxx is the process id */
printf("Filename is %s\n", filename);
```

## Summary

```
#include <stdio.h>

char *gets(buffer);
char *buffer;          /* storage location for input string */
```

## Description

The **gets** function reads a line from the standard input stream **stdin** and stores it in *buffer*. The line consists of all characters up to and including the first newline character ('\n'). The newline character is then replaced with a null character ('\0') before the line is returned.

## Return Value

**Gets** returns its argument. A **NULL** pointer indicates an error or end-of-file condition. Use **ferror** or **feof** to determine whether an error or end-of-file occurred.

## See Also

**fgets, fputs, puts**

## Example

```
#include <stdio.h>

char line[100];
char *result;

/* The following statement gets a line of input from
** stdin.
*/

result = gets(line);
```

## getw

### Summary

```
#include <stdio.h>

int getw(stream);
FILE *stream;          /* pointer to file structure */
```

### Description

The **getw** function reads the next binary value of type **int** from the specified input *stream* and increments the associated file pointer (if there is one) to point to the next unread character. **Getw** does not assume any special alignment of items in the stream.

### Return Value

**Getw** returns the integer value read. A return value of **EOF** may indicate an error or end-of-file; however, the **EOF** value is also a legitimate integer value, so **feof** or **ferror** should be used to verify an end-of-file or error condition.

### See Also

**putw**

---

*Note*

**Getw** is provided primarily for compatibility with previous libraries. Notice that portability problems may occur with **getw** since the size of an **int** and ordering of bytes within an **int** differ across systems.

---

## Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int i;
     .
     .
     .

/* The following statement reads a word from a stream
** and checks for an error.
*/

i = getw(stream);

if (ferror(stream)) {
          perror("getw failed");
          clearerr(stream);
          }
```

# gmtime

## Summary

```
#include <time.h>

struct tm *gmtime(time);
long *time;                    /* pointer to stored time */
```

## Description

The **gmtime** function converts a time stored as a **long** value to a structure. The **long** value *time* represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich Mean Time; this value is usually obtained from a call to **time**.

**Gmtime** breaks down the *time* value and stores it in a structure of type **tm**, defined in *time.h*. The structure result reflects Greenwich Mean Time, not local time.

The fields of the structure type **tm** store the following values.

| Field | Value stored |
|---|---|
| **tm_sec** | Seconds |
| **tm_min** | Minutes |
| **tm_hour** | Hours (0–24) |
| **tm_mday** | Day of month (1–31) |
| **tm_mon** | Month (0–11; January = 0) |
| **tm_year** | Year (current year minus 1900) |
| **tm_wday** | Day of week (0–6; Sunday = 0) |
| **tm_yday** | Day of year (0–365; January 1 = 0) |
| **tm_isdst** | Nonzero if Daylight Savings Time is in effect, otherwise 0 |

Under MS-DOS, dates prior to 1980 are not understood. If *time* represents a date before January 1, 1980, **gmtime** returns the structure representation of 00:00:00 January 1, 1980.

## Return Value

**Gmtime** returns a pointer to the structure result. There is no error return.

## See Also

**asctime, ctime, ftime, localtime, time**

---

*Note*

The **gmtime** and **localtime** functions use a single statically allocated structure to hold the result. Each call to one of these routines destroys the result of the previous call.

---

## Example

```
#include <time.h>

struct tm *newtime;
long ltime;

time(&ltime);
newtime = gmtime(&ltime);
printf("Greenwich Mean Time is %s\n",asctime(newtime));
```

# hypot

## Summary

```
#include <math.h>

double hypot(x,y);
double x, y;          /* floating-point values */
```

## Description

The **hypot** function calculates the length of the hypotenuse, given the length of two sides *x* and *y*. A call to **hypot** is equivalent to:

```
sqrt(x*x + y*y);
```

## Return Value

**Hypot** returns the length of the hypotenuse. If an overflow results, the function sets **errno** to **ERANGE** and returns the value **HUGE**.

## See Also

**cabs**

## Example

```
#include <math.h>

double x, y, z;

x = 3.0;
y = 4.0;

z = hypot(x,y);          /* z = 5.0 */
```

## Summary

```
#include <conio.h>          /* required only for function declarations */

int inp(port);
unsigned port;              /* port number */
```

## Description

The **inp** function reads one byte from the input port specified by *port*. The *port* argument can be any unsigned integer number in the range 0 to 65,535.

## Return Value

**Inp** returns the byte read from *port*. There is no error return.

## See Also

**outp**

## Example

```
#include <conio.h>

unsigned port;
char result;
     .
     .
     .

/* The following statement inputs a byte from the port
** that 'port' is currently set to.
*/

result = inp(port);
```

# int86

## Summary

```
#include <dos.h>

int int86(intno, inregs, outregs);
int intno;                          /* interrupt number */
union REGS *inregs;                 /* register values on call */
union REGS *outregs;                /* register values on return */
```

## Description

The **int86** function executes the 8086 software interrupt specified by the interrupt number *intno*. Before executing the interrupt, **int86** copies the contents of *inregs* to the corresponding registers. After the interrupt returns, the function copies the current register values to *outregs*. It also copies the status of the system carry flag to the **cflag** field in *outregs*. The *inregs* and *outregs* arguments are unions of type **REGS**. The union type is defined in the include file *dos.h*.

**Int86** is intended to be used to invoke DOS interrupts directly.

## Return Value

The return value is the value in the AX register after the interrupt returns. If the **flag** field in *outregs* is nonzero, an error has occurred and the **doserrno** variable is also set to the corresponding error code.

## See Also

**bdos, intdos, intdosx, int86x**

**Example**

```
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

/*
 * Use int86 routine to generate a CONTROL-C interrupt
 * (interrupt number 0x23) which would be caught by the
 * interrupt handling routine int_handler.  Note that the
 * values in the regs struct do not matter for this
 * interrupt.
 */

#define CNTRL_C 0x23
int int_handler(int);
union REGS regs;
          .
          .
          .

signal(SIGINT, int_handler);
          .
          .
          .
int86(CNTRL_C, &regs, &regs);
```

# int86x

## Summary

```
#include <dos.h>

int int86x(intno, inregs, outregs, segregs);
int intno;                          /* interrupt number */
union REGS *inregs;                 /* register values on call */
union REGS *outregs;                /* register values on return */
struct SREGS *segregs;              /* segment register values on call */
```

## Description

The **int86x** function executes the 8086 software interrupt specified by the interrupt number *intno*. Unlike the **int86** function, **int86x** accepts segment register values in *segregs*, letting programs that use long model data segments or far pointers specify which segment or pointer should be used during the system call.

Before executing the specified interrupt, **int86x** copies the contents of *inregs* and *segregs* to the corresponding registers. Only the DS and ES register values in *segregs* are used. After the interrupt returns, the function copies the current register values to *outregs* and restores DS. It also copies the status of the system carry flag to the **cflag** field in *outregs*. The *inregs* and *outregs* arguments are unions of type **REGS**. The *segregs* argument is a structure of type **SREGS**. These types are defined in the include file *dos.h*.

**int86x** is intended to be used to directly invoke DOS interrupts that take an argument in the ES register, or take a DS register value that is different than the default data segment.

## Return Value

The return value is the value in the AX register after the interrupt returns. If the **flag** field in *outregs* is nonzero, an error has occurred and the **doserrno** variable is also set to the corresponding error code.

## See Also

**bdos, intdos, intdosx, int86, segread, FP_SEG**

---

*Note*

Segment values for the *segregs* argument can be obtained by using either the **segread** function or the **FP_SEG** macro.

---

## Example

```
#include <signal.h>
#include <dos.h>
#include <stdio.h>
#include <process.h>

/*
 * Use int86x routine to generate an interrupt 0x21 (system
 * call), which invokes the DOS 'Change Attributes' system
 * call.  The int86x routine is used because the filename to
 * be referenced may be in a segment other than the default
 * data segment (it is referenced by a far pointer), so the
 * DS register must be explicitly set via the SREGS struct.
 */

#define SYSCALL     0x21          /* INT 21H invokes system
                                     calls */
#define CHANGE_ATTR 0x43          /* system call 43H - change
                                     attributes */

char far *filename;               /* filename in 'far' data
                                     segment */

union REGS inregs, outregs;
struct SREGS segregs;
int result;
          .
          .
          .
inregs.h.ah = CHANGE_ATTR;        /* AH is system call
                                     number */
inregs.h.al = 0;                  /* AL is function (get
                                     attributes) */
inregs.x.dx = FP_OFF(filename); /* DS:DX points to file
                                     name */
segregs.ds = FP_SEG(filename);
result = int86x(SYSCALL, &inregs, &outregs, &segregs);
if (outregs.x.cflag) {
    printf("can't get attributes of file; error number %d\n",
            result);
    exit(1);
    }
else {
    printf("Attribs = %#x\n", outregs.x.cx);
    }
```

## Summary

```
#include <dos.h>

int intdos(inregs, outregs);
union REGS *inregs;          /* register values on call */
union REGS *outregs;         /* register values on return */
```

## Description

The **intdos** function invokes the DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. The *inregs* and *outregs* arguments are unions of type **REGS**. The union type is defined in the include file *dos.h*.

To invoke a system call, **intdos** executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* to the corresponding registers. After the INT instruction returns, **intdos** copies the current register values to *outregs*. It also copies the status of the system carry flag to the **cflag** field in *outregs*. If this field is nonzero, the flag was set by the system call and indicates an error condition.

**Intdos** is intended to be used to invoke DOS system calls that take arguments in registers other than DX (DH/DL) and AL, or to invoke system calls that indicate errors by setting the carry flag.

## Return Value

**Intdos** returns the value of the AX register after the system call has completed. If the **flag** field in *outregs* is nonzero, an error has occurred and **doserrno** is also set to the corresponding error code.

## See Also

**bdos, intdosx**

## Example

```
#include <dos.h>
#include <stdio.h>

union REGS inregs, outregs;
          .
          .
          .

/* The following statements get the current date using
** dos function call 2a hex.
*/

inregs.h.ah = 0x2a;
intdos(&inregs,&outregs);
printf("date is %d/%d/%d\n",outregs.h.dh,outregs.h.dl,
          outregs.x.cx);
```

## Summary

```
#include <dos.h>

int intdosx(inregs, outregs, segregs);
union REGS *inregs;                    /* register values on call */
union REGS *outregs;                   /* register values on return */
struct SREGS *segregs;                 /* segment register values on call */
```

## Description

The **intdosx** function invokes the DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. Unlike the **intdos** function, **intdosx** accepts segment register values in *segregs,* letting programs that use long model data segments or far pointers specify which segment or pointer should be used during the system call. The *inregs* and *outregs* arguments are unions of type **REGS**. The *segregs* argument is a structure of type **SREGS**. These types are defined in the include file *dos.h*.

To invoke a system call, **intdosx** executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* and *segregs* to the corresponding registers. Only the DS and ES register values in *segregs* are used. After the INT instruction returns, **intdosx** copies the current register values to *outregs* and restores DS. It also copies the status of the system carry flag to the **cflag** field in *outregs*. If this field is nonzero, the flag was set by the system call and indicates an error condition.

**intdosx** is intended to be used to invoke DOS system calls that take an argument in the ES register, or that take a DS register value that is different from the default data segment.

## Return Value

**intdosx** returns the value of the AX register after the system call has completed. If the **flag** field in *outregs* is nonzero, an error has occurred and **doserrno** is also set to the corresponding error code.

## See Also

**bdos, intdos, segread, FP_SEG**

---

*Note*

Segment values for the *segregs* argument can be obtained by using either the **segread** function or the **FP_SEG** macro.

---

## Example

```
#include <dos.h>

union REGS inregs, outregs;
struct SREGS segregs;
char far *dir = "/test/bin";

/* The following statements change the current working
** directory with dos function call 3b hex.
*/

inregs.h.ah = 0x3b;                    /* change directory */
inregs.x.dx = FP_OFF(dir);             /* file name offset */
segregs.ds = FP_SEG(dir);              /* file name segment */
intdosx(&inregs,&outregs,&segregs);
```

## Summary

```
#include <ctype.h>
int isalnum(c);          /* test for alphanumeric ('A'–'Z', 'a'–'z', or '0'–'9') */
int isalpha(c);          /* test for letter ('A'–'Z' or 'a'–'z') */
int isascii(c);          /* test for ASCII character (0x00–0x7F) */
int c;                   /* integer to be tested */
```

## Description

The **ctype** routines listed above test a given integer value, returning a nonzero value if the integer satisfies the test condition and a zero value if it does not. An ASCII character set environment is assumed.

The **isascii** routine produces a meaningful result for all integer values. However, the remaining routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where **isascii** holds true) or for the non-ASCII value **EOF** (defined in *stdio.h*).

## See Also

**iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, toascii, tolower, toupper**

---

*Note*

The **ctype** routines are implemented as macros.

---

## Example

```
#include <stdio.h>
#include <ctype.h>

int ch;

/* The following statements analyze all characters
** between code 0x0 and code 0x7f, printing "A" for
** alphas, "AN" for alphanumerics, and "AS" for asciis.
*/

for (ch = 0; ch <= 0x7f; ch++) {
        printf("%#04x",ch);
        printf("%3s",isalnum(ch)  ? "AN" : "");
        printf("%2s",isalpha(ch)  ? "A"  : "");
        printf("%3s",isascii(ch)  ? "AS" : "");

putchar('\n');
}
```

## Summary

```
#include <io.h>          /* required only for function declarations */

int isatty(handle);
int handle;              /* handle referring to device to be tested */
```

## Description

The **isatty** function determines whether the given *handle* is associated with a character device (that is, a terminal, console, printer or serial port).

## Return Value

**Isatty** returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

## Example

```
#include <io.h>

int fh;
long loc;
          .
          .
          .
if (isatty(fh) == 0)
        loc = tell(fh);    /* if not a device, get current
                           ** position
                           */
```

## Summary

```
#include <ctype.h>

iscntrl(c);      /* test for control character (0x00–0x1f or 0x7f) */

isdigit(c);      /* test for digit ('0'–'9') */

isgraph(c);      /* test for printable character not including the space */
                 /* character (0x21–0x7e) */

islower(c);      /* test for lower case ('a'–'z') */

isprint(c);      /* test for printable character (0x20–0x7e) */

ispunct(c);      /* test for punctuation character (isalnum(c) */
                 /* and iscntrl(c) both false) */

isspace(c);      /* test for whitespace character (0x09–0x0d or 0x20) */

isupper(c);      /* test for upper case ('A'–'Z') */

isxdigit(c);     /* test for hexadecimal digit ('A'–'F', 'a'–'f', or '0'–'9') */

int c;           /* integer value to be tested */
```

## Description

The **ctype** routines listed above test a given integer value, returning a nonzero value if the integer satisfies the test condition, and zero if it does not. An ASCII character set environment is assumed.

These routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where **isascii** holds true) or for the non-ASCII value **EOF** (defined in *stdio.h*).

## See Also

isalnum, isalpha, isascii, toascii, tolower, toupper

---

*Note*

The **ctype** routines are implemented as macros.

---

## Example

```
#include <stdio.h>
#include <ctype.h>

int ch;

/* The following statements analyze all characters
** between code 0x0 and code 0x7f, printing "U" for
** uppers, "L" for lowers, "D" for digits, "X" for
** hex digits, "S" for spaces, "PU" for punctuations,
** "PR" for printables, "G" for graphics, and "C" for
** controls.  If the code is printable, it is
** printed.
*/

for (ch = 0; ch <= 0x7f; ch++) {
        printf("%2s",iscntrl(ch)  ? "C"  : "");
        printf("%2s",isdigit(ch)  ? "D"  : "");
        printf("%2s",isgraph(ch)  ? "G"  : "");
        printf("%2s",islower(ch)  ? "L"  : "");
        printf(" %c",isprint(ch)  ? ch   : '\0');
        printf("%3s",ispunct(ch)  ? "PU" : "");
        printf("%2s",isspace(ch)  ? "S"  : "");
        printf("%3s",isprint(ch)  ? "PR" : "");
        printf("%2s",isupper(ch)  ? "U"  : "");
        printf("%2s",isxdigit(ch) ? "X"  : "");

putchar('\n');
}
```

# itoa

## Summary

```
#include <stdlib.h>              /* required only for function declarations */

char *itoa(value, string, radix);
int value;                       /* number to be converted */
char *string;                    /* string result */
int radix;                       /* base of value */
```

## Description

The **itoa** function converts the digits of the given *value* to a null-terminated character string and stores the result in *string*. The *radix* argument specifies the base of *value;* it must be in the range 2–36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

## Return Value

**itoa** returns a pointer to *string*. There is no error return.

## See Also

**itoa, ultoa**

---

*Note*

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 17 bytes.

---

## Example

```
#include <stdlib.h>

int radix = 8;
char buffer[20];
char *p;
          .
          .
          .
p = itoa(-3445,buffer,radix);     /* p = "171213" */
```

## kbhit

### Summary

```
#include <conio.h>   /* required only for function declarations */
int kbhit( );
```

### Description

The **kbhit** function checks the console for a recent keystroke.

### Return Value

**Kbhit** returns a nonzero value if a key has been pressed. Otherwise, it returns zero.

### Example

```
#include <conio.h>

int result;

/* The following statement tests to see if a key has
** been hit.
*/

result = kbhit();

/* If result is nonzero, a keystroke is waiting in the
** buffer. It can be fetched with getch or getche.
** If getch or getche were called without first checking
** kbhit, the program might pause while waiting for
** input.
*/
```

## Summary

```
#include <stdlib.h>    /* required only for function declarations */

long labs(n);
long n;                /* long integer value */
```

## Description

The **labs** function produces the absolute value of its long integer argument *n*.

## Return Value

**Labs** returns the absolute value of its argument. There is no error return.

## See Also

**abs, cabs, fabs**

## Example

```
#include <stdlib.h>

long x, y;

x = -41567L;
y = labs(x);      /* y = 41567L */
```

# ldexp

## Summary

```
#include <math.h>

double ldexp(x, exp);
double x;                    /* floating-point value */
int exp;                     /* integer exponent */
```

## Description

The **ldexp** function calculates the value of $x * 2^{exp}$.

## Return Value

**Ldexp** returns $x * 2^{exp}$. If an overflow results, the function returns positive or negative **HUGE** (the sign depends on the sign of *value*). If an underflow results, the function returns 0. In both cases, **errno** is set to **ERANGE**.

## See Also

**frexp, modf**

## Example

```
#include <math.h>

double x, y;
int p;

x = 1.5;
p = 5;
y = ldexp(x,p);              /* y = 48.0 */
```

## Summary

```
#include <time.h>

struct tm *localtime(time);
long *time;                    /* pointer to stored time */
```

## Description

The **localtime** function converts a time stored as a **long** value to a structure. The **long** value *time* represents the seconds elapsed since 00:00:00, January 1, 1970, Greenwich Mean Time; this value is usually obtained from the *time* function.

**Localtime** breaks down the *time* value, corrects for the local time zone and Daylight Savings Time if appropriate, and stores the corrected time in a structure of type *tm*. (See **gmtime** for a description of the *tm* structure fields).

Under MS-DOS, dates prior to 1980 are not understood. If *time* represents a date before January 1, 1980, **localtime** returns the structure representation of 00:00:00 January 1, 1980.

**Localtime** makes corrections for the local time zone if the user first sets the environment variable TZ. The value of TZ must be a three-letter time zone name (such as PST), followed by a possibly signed number giving the difference between Greenwich Mean Time and the local time zone. The number may be followed by a three-letter Daylight Savings Time zone (such as PDT). **Localtime** uses the difference between Greenwich Mean Time and local time to adjust the stored time value. If a Daylight Savings Time zone is present in the TZ setting, **localtime** also corrects for daylight savings time. If TZ currently has no value, the default value PST8PDT is used.

When TZ is set, three other environment variables, **timezone**, **daylight**, and **tzname**, are automatically set as well. See the **tzset** function for a description of these variables.

## Return Value

**Localtime** returns a pointer to the structure result. There is no error return.

## See Also

**asctime, ctime, ftime, gmtime, time, tzset**

---

*Note*

The **gmtime** and **localtime** functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

---

## Example

```
#include <time.h>
#include <stdio.h>

struct tm *newtime;
long ltime;

time(&ltime);
newtime = localtime(&ltime);
printf("the time is %s\n",asctime(newtime));
```

## Summary

```
#include <sys\locking.h>
#include <io.h>                    /* required only for function declarations */

int locking(handle, mode, nbyte);
int handle;                        /* file handle */
int mode;                          /* file locking mode */
long nbyte;                        /* number of bytes to lock */
```

## Description

The **locking** function locks or unlocks *nbyte* bytes of the file specified by *handle*. Locking bytes in a file prevents subsequent reading and writing of those bytes by other processes. Unlocking a file permits other processes to read or write to previously locked bytes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbyte* bytes, or to the end of the file.

*Mode* specifies the locking action be performed. It must be one of the following manifest constants.

| Manifest Constant | Meaning |
|---|---|
| LK_LOCK | Lock the specified bytes. If the bytes cannot be locked, try again after 1 second. If after 10 attempts, the bytes cannot be locked, return an error. |
| LK_RLCK | Same as LK_LOCK. |
| LK_NBLCK | Lock the specified bytes. If bytes cannot be locked, return an error. |
| LK_NBRLCK | Same as LK_NBLCK. |
| LK_UNLCK | Unlock the specified bytes. The bytes must have been previously locked. |

More than one region of a file can be locked, but no overlapping regions are allowed. Furthermore, no more than one region can be unlocked at a time.

When unlocking a file, the region of the file being unlocked must correspond to a region that was previously locked. The **locking** function does not coalesce adjacent regions, so if two locked regions are adjacent, each region must be unlocked separately.

All locks should be removed before closing a file or exiting the program.

## Return Value

**Locking** returns 0 if it is successful. A return value of −1 indicates failure, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Locking violation (file already locked or unlocked). |
| EBADF | Invalid file handle. |
| EDEADLOCK | Locking violation: returned when the **LK_LOCK** or **LK_RLCK** flag is specified and the file cannot be locked after 10 attempts. |
| EINVAL | SHARE.COM not installed. |

## See Also

creat, open

---

*Note*

**Locking** should be used only under MS-DOS 3.0 and later; it has no effect under earlier versions of MS-DOS.

---

## Example

```
#include <io.h>
#include <sys\locking.h>
#include <stdlib.h>

extern unsigned char _osmajor;
int fh;
long pos;
          .
          .
          .
                    /* save the current file pointer position,
                    ** then lock a region from the beginning of
                    ** the file to the saved file pointer
                    ** position
                    */
if (_osmajor >= 3)  {
          pos = tell(fh);
          lseek(fh, 0L, 0);
          if ( (locking(fh, LK_NBLCK, pos)) != -1) {
                              .
                              .
                              .
                    lseek(fh, 0L, 0);
                    locking(fh, LK_UNLCK, pos);
                    }
          }
```

# log – log10

## Summary

```
#include <math.h>

double log(x);          /* calculate natural logarithm of x */

double log10(x);        /* calculate logarithm base 10 of x */

double x;               /* floating-point value */
```

## Description

The **log** and **log10** functions calculate the natural logarithm and base 10 logarithm of $x$, respectively.

## Return Value

**Log** and **log10** return the logarithm result. If $x$ is negative, both functions print a **DOMAIN** error message to **stderr** and return the value negative **HUGE**. If $x$ is zero, both functions print a **SING** error message and return the value negative **HUGE**. In either case, **errno** is set to **EDOM**.

Error handling can be modified by using the **matherr** routine.

## See Also

**exp, matherr, pow**

## Example

```
#include <math.h>

double x = 1000.0, y;

y = log(x);       /* y = 6.907755 */

/* Log10 calculates the base 10 logarithm of the given
** value.
*/
y = log10(x);     /* y = 3.0 */
```

## Summary

```
#include <setjmp.h>

void longjmp(env, value);
jmp_buf env;                    /* variable in which environment is stored */
int value;                      /* value to be returned to setjmp call */
```

## Description

The **longjmp** function restores a stack environment previously saved in *env* by **setjmp**. **Setjmp** and **longjmp** provide a way to execute a nonlocal goto and are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to **setjmp** causes the current stack environment to be saved in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point just after the corresponding **setjmp** call. Execution resumes as if the given *value* had just been returned by the **setjmp** call. The values of all variables (except register variables) accessible to the routine receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable.

**Longjmp** must be called before the function that called **setjmp** returns. Calling **longjmp** after the function calling **setjmp** returns will cause unpredictable program behavior.

The *value* returned by **longjmp** must be nonzero. If a zero argument is given for *value*, the value 1 is substituted in the actual return.

## Return Value

**Longjmp** does not return a value.

## See Also

**setjmp**

---

*Warning*

The values of register variables in the routine calling **setjmp** may not be restored to the proper values after a **longjmp** is executed.

---

## Example

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

main()
{
        if (setjmp(mark) != 0) {
                printf("longjmp has been called\n");
                recover();
                exit(1);
                }
        printf("setjmp has been called\n");
                .
                .
                .
        p();
                .
                .
                .
}

p()
{
        int error = 0;
                .
                .
                .
        if (error != 0)
                longjmp(mark,-1);
                .
                .
                .
}

recover()
{
        /* ensure that data files won't be corrupted by
        ** exiting the program
        */
                .
                .
                .
}
```

# lseek

## Summary

```
#include <io.h>                    /* required only for function declarations */

long lseek(handle, offset, origin);
int handle;                        /* handle referring to open file */
long offset;                       /* number of bytes from origin */
int origin;                        /* initial position */
```

## Description

The **lseek** function moves the file pointer (if any) associated with *handle* to a new location that is *offset* bytes from the *origin*. The next operation on the file takes place at the new location. *Origin* must be one of the following constants.

| Origin | Definition |
|---|---|
| 0 | Beginning of file |
| 1 | Current position of file pointer |
| 2 | End of file |

**Lseek** can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. However, an attempt to position the pointer before the beginning of the file causes an error.

## Return Value

**Lseek** returns the offset, in bytes, of the new position from the beginning of the file. A return value of −1L indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EBADF | Invalid file handle. |
| EINVAL | Invalid value for *origin*, or position specified by *offset* is before the beginning of the file. |

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

## See Also

**fseek, tell**

## Example

```
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>

int fh;
long position;

fh = open("data",O_RDONLY);
          .
          .
          .
/* 0 offset from beginning */

position = lseek(fh,0L,0);
if (position == -1L)
          perror("lseek to beginning failed");
          .
          .
          .
/* find current position */
position = lseek(fh,0L,1);
if (position == -1L)
          perror("lseek to current position failed");
          .
          .
          .
/* go to end of file */
position = lseek(fh,0L,2);
if (position == -1L)
          perror("lseek to end failed");
```

# ltoa

## Summary

```
#include <stdlib.h>          /* required only for function declarations */

char *ltoa(value, string, radix);
long value;                  /* number to be converted */
char *string;                /* string result */
int radix;                   /* base of value */
```

## Description

The **ltoa** function converts the digits of the given *value* to a null-terminated character string and stores the result in *string*. The *radix* argument specifies the base of *value;* it must be in the range 2-36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

## Return Value

**Ltoa** returns a pointer to *string*. There is no error return.

## See Also

**itoa, ultoa**

---

*Note*

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes.

---

## Example

```
#include <stdlib.h>

int radix = 10;
char buffer[20];
char *p;

p = ltoa(-344115L,buffer,radix);  /* p = "-344115" */
```

# malloc

## Summary

```
#include <malloc.h>  /* required only for function declarations */

char *malloc(size);
unsigned size;          /* bytes in allocated block */
```

## Description

The **malloc** function allocates a memory block of at least *size* bytes. (The block may be larger than *size* bytes due to space required for alignment and for maintenance information.)

## Return Value

**Malloc** returns a **char** pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **char**, use a type cast on the return value. The return value is **NULL** if there is insufficient memory available.

## See Also

**calloc, free, realloc**

## Example

```
#include <malloc.h>

int *intarray;

/* Allocate space for 20 integers */

intarray = (int *)malloc(20*sizeof(int));
```

## Summary

```
#include <math.h>

int matherr(x);
struct exception *x;        /* math exception information */
```

## Description

The **matherr** function processes errors generated by the functions of the math library. The math functions call **matherr** whenever an error is detected. The user can provide a different definition of the **matherr** function to carry out special error-handling.

When an error occurs in a math routine, **matherr** is called with a pointer to the following structure (defined in *math.h*) as an argument.

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
        };
```

*Type* specifies the type of math error. It will be one of the following values, defined in *math.h*.

| Value | Meaning |
|---|---|
| DOMAIN | Argument domain error |
| SING | Argument singularity |
| OVERFLOW | Overflow range error |
| UNDERFLOW | Underflow range error |
| TLOSS | Total loss of significance |
| PLOSS | Partial loss of significance |

**Name** is a pointer to a null-terminated string containing the name of the function that caused the error. **Arg1** and **arg2** specify the argument values that caused the error. (If only one argument is given, it is stored in **arg1**.)

**Retval** is the default return value for this error; the user may change the return value. **Matherr**'s return value must specify whether or not an error actually occurred. If **matherr** returns zero, an error message is displayed and **errno** is set to an appropriate error value. If **matherr** returns a nonzero value, no error message is displayed and **errno** remains unchanged.

## Return Value

**Matherr** should return zero to indicate an error, and non-zero to indicate successful corrective action.

## See Also

**acos, asin, atan, atan2, bessel, cabs, cos, cosh, exp, hypot, log, pow, sin, sinh, sqrt, tan**

## Example

```
#include <math.h>
#include <string.h>

/*
 * Catches errors in calls to the log or log10 routines.  If
 * the error is the result of a negative argument (DOMAIN
 * error), the log or log10 of the absolute value of the
 * argument is returned (rather than the default value, HUGE).
 * The error message is suppressed.  If the error is a zero
 * argument, or the error was generated by some other routine,
 * the default actions are taken.
 */

int matherr(x)
struct exception *x;
{
        if (x->type == DOMAIN) {
                if (strcmp(x->name, "log") == 0) {
                        x->retval = log(-(x->arg1));
                        return(1);
                        }
                else if (strcmp(x->name, "log10") == 0) {
                        x->retval = log10(-(x->arg1));
                        return(1);
                        }
                }
        return(0);                      /* use default actions */
}
```

# memccpy

## Summary

```
#include <memory.h>                    /* required only for function declarations */

char *memccpy (dest, src, c, cnt);
char *dest;                            /* pointer to destination */
char *src;                             /* pointer to source */
char c;                                /* last character to copy */
unsigned cnt;                          /* number of characters */
```

## Description

The **memccpy** function copies zero or more bytes of *src* to *dest*, copying up to and including the first occurrence of the character *c* or until *cnt* bytes have been copied, whichever comes first.

## Return Value

If the character *c* is copied, **memccpy** returns a pointer to the byte in *dest* that immediately follows the character. If *c* is not copied, **memccpy** returns **NULL**.

## See Also

**memchr, memcmp, memcpy, memset**

## Example

```
#include <memory.h>

char buffer[100], source[100];
char *result;
          .
          .
          .
/* Copy bytes from source to buffer until '\n' is
** copied, but not more than 100 bytes.
*/

result = memccpy(buffer,source,'\n',100);
```

## Summary

```
#include <memory.h>          /* required only for function declarations */

char *memchr(buf, c, cnt);
char *buf;                   /* pointer to buffer */
char c;                      /* character to copy */
unsigned cnt;                /* number of characters */
```

## Description

The **memchr** function searches the first *count* bytes of *buf* for the first occurrence of the character *c*. The search continues until *c* is found or *cnt* bytes have been examined.

## Return Value

**Memchr** returns a pointer to the location of *c* in *buf*. It returns **NULL** if *c* is not within the first *cnt* bytes of *buf*.

## See Also

**memccpy, memcmp, memcpy, memset**

## Example

```
#include <memory.h>

char buffer[100];
char *result;
          .
          .
          .
/* Find the first occurrence of 'a' in buffer. If 'a' is
** not in the first 100 bytes, return NULL.
*/

result = memchr(buffer,'a',100);
```

## memcmp

### Summary

```
#include <memory.h>          /* required only for function declarations */

int memcmp (buf1, buf2, cnt);
char *buf1;                  /* first buffer */
char *buf2;                  /* second buffer */
unsigned cnt;                /* number of characters */
```

### Description

The **memcmp** function compares the first *cnt* bytes of *buf1* and *buf2* lexicographically and returns a value indicating their relationship, as follows.

| Value | Meaning |
|---|---|
| Less than 0 | *buf1* less than *buf2* |
| 0 | *buf1* identical to *buf2* |
| Greater than 0 | *buf1* greater than *buf2* |

### Return Value

**Memcmp** returns an integer value as described above.

### See Also

**memccpy, memchr, memcpy, memset**

## Example

```
#include <memory.h>

char first[100], second[100];
int result;
          .
          .
          .
/* The following statement compares first[] and
** second[] to see which, if either, is greater. If
** they are the same in the first 100 bytes, they are
** considered equal.
*/

result = memcmp(first,second,100);
```

# memcpy

## Summary

```
#include <memory.h>          /* required only for function declarations */

char *memcpy(dest, src, cnt);
char *dest;                  /* pointer to destination */
char *src;                   /* pointer to source */
unsigned cnt;                /* number of characters */
```

## Description

The **memcpy** function copies *cnt* bytes of *src* to *dest*. If some regions of *src* and *dest* overlap, **memcpy** ensures that the original *src* bytes in the overlapping region are copied before being overwritten.

## Return Value

**Memcpy** returns a pointer to *dest*.

## See Also

**memccpy, memchr, memcmp, memset**

## Example

```
#include <memory.h>

char source[200], destination[200];
          .
          .
          .
/* Move 200 bytes from source to destination, and
** return a pointer to destination.
*/

memcpy(destination,source,200);
```

## Summary

```
#include <memory.h>          /* required only for function declarations */

char *memset (dest, c, cnt);
char *dest;                  /* pointer to destination */
char c;                      /* character to set */
unsigned cnt;                /* number of characters */
```

## Description

The **memset** function sets the first *cnt* bytes of *dest* to the character *c*.

## Return Value

**Memset** returns a pointer to *dest*.

## See Also

**memccpy, memchr, memcmp, memcpy**

## Example

```
#include <memory.h>

char buffer[100];

/* Set the first 100 bytes of buffer to zeros.
*/

memset(buffer,'\0',100);
```

# mkdir

## Summary

```
#include <direct.h>          /* required only for function declarations */

int mkdir(pathname);
char *pathname;              /* pathname for new directory */
```

## Description

The **mkdir** function creates a new directory with the specified *pathname*. Only one directory can be created at a time, so only the last component of *pathname* can name a new directory.

## Return Value

**Mkdir** returns the value 0 if the new directory was created. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Directory not created: the given name is the name of an existing file, directory, or device. |
| ENOENT | Pathname not found. |

## See Also

chdir, rmdir

## Example

```
#include <direct.h>

int result;

/* The following two statements create two new directories:
** one at the root on drive b:, and one in the "tmp"
** subdirectory of the current working directory.
*/

result = mkdir("b:/tmp");          /* "b:\\tmp" could also
                                   ** be used
                                   */
   .
   .
   .
result = mkdir("tmp/sub");         /* "tmp\\sub" could also
                                   ** be used
                                   */
```

# mktemp

## Summary

```
#include <io.h>                    /* required only for function declarations */

char *mktemp(template);
char *template;                    /* filename pattern */
```

## Description

The **mktemp** function creates a unique filename by modifying the given *template*. The *template* argument has the form

*base*XXXXXX

where *base* is the part of the new filename supplied by the user and the X's are placeholders for the part supplied by **mktemp**. **Mktemp** preserves *base* and replaces the six trailing X's with an alphanumeric character followed by a five-digit value. The five-digit value is a unique number identifying the calling process. The alphanumeric character is zero ('0') the first time **mktemp** is called with a given *template*.

In subsequent calls from the same process with the same *template*, **mktemp** checks to see whether previously returned names have been used to create files. If no file exists for a given name, **mktemp** returns that name. If files exist for all previously returned names, **mktemp** creates a new name by replacing the alphanumeric character in the name with the next available lowercase letter. For example, if the first name returned is "t012345" and this name is used to create a file, the next name returned will be "ta12345." When creating new names **mktemp** uses, in order, '0' and the lowercase letters 'a' to 'z'.

## Return Value

**Mktemp** returns a pointer to the modified *template*. The return value is **NULL** if the *template* argument is badly formed or no more unique names can be created from this *template*.

## See Also

**fopen, getpid, open**

---

*Note*

**Mktemp** generates unique filenames but does not create or open files.

---

## Example

```
#include <io.h>

char *template = "fnXXXXXX";
char *result;

/* The following statement calls mktemp to generate a unique
** filename.
*/

result = mktemp(template);
```

# modf

## Summary

```
#include <math.h>

double modf(x, intptr);
double x;                    /* floating-point value */
double *intptr;              /* pointer to stored integer portion */
```

## Description

The **modf** function breaks down the floating-point value *x* into fractional and integer parts. The signed fractional portion of *x* is returned. The integer portion is stored as a floating-point value at *intptr*.

## Return Value

**Modf** returns the signed fractional portion of *x*. There is no error return.

## See Also

**frexp, ldexp**

## Example

```
#include <math.h>

double x, y, n;

x = -14.87654321;
y = modf(x,&n);          /* y = -0.87654321, n = -14.0 */
```

## Summary

```
#include <memory.h>          /* required only for function declarations */

void movedata(srcseg, srcoff, destseg, destoff, nbytes);
int srcseg;                  /* segment address of source */
int srcoff;                  /* segment offset of source */
int destseg;                 /* segment address of destination */
int destoff;                 /* segment offset of destination */
unsigned nbytes;             /* number of bytes */
```

## Description

The **movedata** function copies *nbytes* bytes from the source address specified by *srcseg*:*srcoff* to the destination address specified by *destseg*:*destoff*.

**Movedata** is intended to be used to move far data in small or medium model programs where segment addresses of data are not implicitly known. In large model programs, the **memcpy** function can be used since segment addresses are implicitly known.

## Return Value

There is no return value.

## See Also

**memcpy, segread, FP_SEG**

---

*Note*

Segment values for the *srcseg* and *destseg* arguments can be obtained by using either the **segread** function or the **FP_SEG** macro.

**Movedata** does not handle all cases of overlapping moves correctly (overlapping moves occur when part of the destination is the same memory area as part of the source). Overlapping moves are handled correctly in the **memcpy** function.

---

## Example

```
#include <memory.h>
#include <dos.h>

char far *src;
char far *dest;
          .
          .
          .
/* The following statement moves 512 bytes of data from
** src to dest.
*/

movedata(FP_SEG(src),FP_OFF(src),FP_SEG(dest),
         FP_OFF(dest),512);
```

## Summary

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>                    /* required only for function declarations */

int open(pathname, oflag [, pmode ]);
char *pathname;                    /* file pathname */
int oflag;                         /* type of operations allowed */
int pmode;                         /* permission setting */
```

## Description

The **open** function opens the file specified by *pathname* and prepares the file for subsequent reading or writing as defined by *oflag*. *Oflag* is an integer expression formed by combining one or more of the following manifest constants, defined in *fcntl.h*. When more than one manifest constant is given, the constants are joined with the bitwise OR operator (|).

| Oflag | Meaning |
|---|---|
| O_APPEND | Reposition the file pointer to the end of the file before every write operation. |
| O_CREAT | Create and open a new file for writing; this has no effect if the file specified by *pathname* exists. |
| O_EXCL | Return an error value if the file specified by *pathname* exists. Only applies when used with O_CREAT. |
| O_RDONLY | Open file for reading only; if this flag is given, neither O_RDWR nor O_WRONLY may be given. |
| O_RDWR | Open file for both reading and writing; if this flag is given, neither O_RDONLY nor O_WRONLY may be given. |
| O_TRUNC | Open and truncate an existing file to 0 length; the file must have write permission, and the contents of the file are destroyed. |
| O_WRONLY | Open file for writing only; if this flag is given, neither O_RDONLY nor O_RDWR may be given. |
| O_BINARY | Open file in binary (untranslated) mode. (See *fopen* for a description of binary mode.) |
| O_TEXT | Open file in text (translated) mode. (See *fopen* for a description of text mode.) |

---

*Note*

O_TRUNC destroys the complete contents of an existing file. Use with care.

---

The *pmode* argument is required only when O_CREAT is specified. If the file exists, *pmode* is ignored. Otherwise, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. The *pmode* is an integer expression containing one or both of the manifest constants S_IWRITE and S_IREAD, defined in *sys\stat.h*. When both constants are given, they are joined with the bitwise OR operator (|). The meaning of the *pmode* argument is as follows.

| Value | Meaning |
|---|---|
| S_IWRITE | Writing permitted |
| S_IREAD | Reading permitted |
| S_IREAD \| S_IWRITE | Reading and writing permitted |

If write permission is not given, the file is read-only. Under MS-DOS all files are readable; it is not possible to give write-only permission. Thus, the modes **S_IWRITE** and **S_IREAD | S_IWRITE** are equivalent.

**Open** applies the current file permission mask to *pmode* before setting the permissions (see **umask**).

## Return Value

**Open** returns a file handle for the opened file. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Given pathname is a directory; or the file is read-only but an open for writing was attempted; or a sharing violation occurred (the file's sharing mode does not allow the specified operations; MS-DOS version 3.0 or later only). |
| EEXIST | The **O_CREAT** and **O_EXCL** flags are specified but the named file already exists. |
| EMFILE | No more file handles available (too many open files). |
| ENOENT | File or pathname not found. |

## See Also

**access, chmod, close, creat, dup, dup2, fopen, sopen, umask**

## Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>

int fh1, fh2;

fh1 = open("data1",O_RDONLY);
if (fh1 == -1)
        perror("open failed on input file");

fh2 = open("data2",O_WRONLY|O_TRUNC|O_CREAT,S_IREAD|S_IWRITE);
if (fh2 == -1)
        perror("open failed on output file");
```

## Summary

```
#include <conio.h>   /* required only for function declarations */

int outp(port, value);
unsigned port;       /* port number */
int value;           /* output value */
```

## Description

The **outp** function writes the specified *value* to the output port specified by *port*. The *port* argument can be any unsigned integer in the range 0 to 65,535; *value* can be any integer in the range 0 to 255.

## Return Value

**Outp** returns *value*. There is no error return.

## See Also

**inp**

## Example

```
#include <conio.h>
int port, byte-val;
          .
          .
          .
/* The following statement outputs a byte to the port
** that 'port' is currently set to.
*/

outp(port,byte-val);
```

# perror

## Summary

```
#include <stdlib.h>                    /* required only for function declarations */

void perror(string);
char *string;                          /* user-supplied message */

int errno;                             /* error number */
int sys_nerr;                          /* number of system messages */
char *sys_errlist[sys_nerr];           /* array of error messages */
```

## Description

The **perror** function prints an error message to **stderr**. The *string* argument is printed first, followed by a colon, the system error message for the last library call that produced an error, and a newline.

The actual error number is stored in the variable **errno**, which should be declared at the external level. The system error messages are accessed through the variable **sys_errlist**, which is an array of messages ordered by error number. **Perror** prints the appropriate error message by using the **errno** value as an index to **sys_errlist**. The value of the variable **sys_nerr** is defined as the maximum number of elements in the **sys_errlist** array.

To produce accurate results, **perror** should be called immediately after a library routine returns with an error. Otherwise, the **errno** value may be overwritten by subsequent calls.

## Return Value

**Perror** returns no value.

## See Also

clearerr, ferror

---

*Note*

Under MS-DOS, some of the **errno** values listed in *errno.h* are not used. See Appendix A, "Error Messages," for a list of **errno** values used on MS-DOS and the corresponding error messages. The **perror** function prints an empty string for any **errno** value not used under MS-DOS.

---

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <process.h>

FILE *stream;

/* The following statements attempt to open a stream. If
** the open fails, a message is printed and the program
** aborts.
*/

if ((stream = fopen("data","r")) == NULL) {
        perror("couldn't open data file");
        abort();
        }
```

# pow

## Summary

```
#include <math.h>

double pow(x,y);
double x;              /* number to be raised */
double y;              /* power of x */
```

## Description

The **pow** function computes $x$ raised to the $y$th power.

## Return Value

**Pow** returns the value of $x^y$. If $x$ is zero and $y$ is nonpositive or if $x$ is negative and $y$ is not an integer, the function prints a **DOMAIN** error message to **stderr**, sets **errno** to **EDOM**, and returns zero. If an overflow results, the function sets **errno** to **ERANGE** and returns either positive or negative **HUGE**. If an underflow results, the function sets **errno** to **ERANGE** and returns 0. No message is printed for overflow or underflow conditions.

## See Also

**exp, log, sqrt**

## Example

```
#include <math.h>
double x = 2.0, y = 3.0, z;
          .
          .
          .
z = pow(x,y);     /* z = 8.0 */
```

## Summary

```
#include <stdio.h>

int printf(format-string [, argument... ]);
char *format-string;                    /* format control */
```

## Description

The **printf** function formats and prints a series of characters and values to the standard output stream **stdout**. The *format-string* consists of ordinary characters and format specifications. The ordinary characters are simply copied in order of their appearance to **stdout**. Format specifications, beginning with a percent sign (%), determine the output format for the *arguments*, if any, following the *format-string*.

The *format-string* is read left to right. When the first format specification (if any) is encountered, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* to be converted and output, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

A format specification has the following form

%[*flags*] [*width*] [.*precision*] [**l**] *type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, "%s"). The optional fields control other aspects of the formatting, as follows.

| Field | Description |
|---|---|
| *flags* | Justification of output and printing of signs, blanks, decimal points, octal and hexadecimal prefixes |
| *width* | Minimum number of characters output |
| *precision* | Maximum number of characters printed for all or part of the output field, or minimum number of digits printed for integer values |
| l | Size of argument expected |

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to **stdout**. For example, to print a percent sign character, use "%%".

The *type* characters and their meanings are as follows.

| Character | Argument Type | Output Format |
|---|---|---|
| d | Integer | Signed decimal integer. |
| u | Integer | Unsigned decimal integer. |
| o | Integer | Unsigned octal integer. |
| x | Integer | Unsigned hexadecimal integer, using "abcdef". |
| X | Integer | Unsigned hexadecimal integer, using "ABCDEF". |
| f | Floating-point | Signed value having the form [−] *dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision. |
| e | Floating-point | Signed value having the form [−] *d.dddd* **e** [*sign*] *ddd*, where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or −. |
| E | Floating-point | Identical to the "e" format except that "E" introduces the exponent instead of "e". |
| g | Floating-point | Signed value printed in "f" or "e" format, whichever is more compact for the given value and *precision* (see below). The "e" format is used only when the exponent of the value is less than −4 or greater than *precision*. Trailing zeroes are truncated and the decimal point appears only if one or more digits follow it. |
| G | Floating-point | Identical to the "g" format except that "E" introduces the exponent (where appropriate) instead of "e". |
| c | Character | Single character. |
| s | String | Characters printed up to the first null character ('\0') or until *precision* is reached. |
| l | Long integer | Used as a prefix with **d**, **o**, **u**, **x**, and **X** types to specify an argument having **long** type. |

The *flag* characters and their meanings are as follows (notice that more than one *flag* can appear in a format specification).

| Flag | Meaning | Default |
|---|---|---|
| – | Left-justify the result within the field *width.* | Right-justify. |
| + | Prefix the output value with a sign (+ or –) if the output value is of a signed type. | Sign appears only for negative signed values (–). |
| *blank* (' ') | Prefix the output value with a blank if the output value is signed and positive. The "+" flag overrides the *blank* flag if both appear, and a positive signed value will be output with a sign. | No blank. |
| # | When used with the **o**, **x**, or **X** formats, the "#" flag prefixes any nonzero output value with 0, 0x, or 0X, respectively. | No prefix. |
| | When used with the **f**, **e**, or **E** formats, the "#" flag forces the output value to contain a decimal point in all cases. | Decimal point appears only if digits follow it. |
| | When used with the **g** or **G** formats, the "#" flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. | Decimal point appears only if digits follow it; trailing zeros are truncated. |
| | Ignored when used with **c**, **d**, **u**, or **s**. | |

*Width* is a non-negative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width,* blanks are added on the left or the right (depending on whether the "–" flag is specified) until the minimum width is reached. If *width* is prefixed with a zero ('0'), zeroes are added until the minimum width is reached (not useful for left-justified numbers).

*Width* never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width,* or *width* is not given, all characters of the value are printed (subject to the *precision* specification).

The *width* specification may be an asterisk (*), in which case an argument from the argument-list supplies the value. The *width* argument must precede the value being formatted in the argument list.

*Precision* is a non-negative decimal integer preceded by a period (.), which specifies the number of characters to be printed, or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value, or rounding in the case of a floating-point value.

The *precision* specification may be an asterisk (*), in which case an argument from the argument-list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as follows.

| Type | Meaning | Default |
|---|---|---|
| d<br>u<br>o<br>x<br>X | *Precision* specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*. | If *precision* is 0 or omitted entirely, or if the period (.) appears without a number following it, the *precision* is set to 1. |
| f<br>e<br>E | *Precision* specifies the number of digits to be printed after the decimal point. The last digit printed is rounded. | Default *precision* is six. If *precision* is zero or the period (.) appears without a number following it, no decimal point is printed. |
| g<br>G | *Precision* specifies the maximum number of significant digits printed. | All significant digits are printed. |
| c | No effect. | Character printed. |
| s | *Precision* specifies the maximum number of characters to be printed. Characters in excess of *precision* are not printed. | Characters are printed until a null character is encountered. |

## Return Value

Printf returns the number of characters printed.

## See Also

fprintf, scanf, sprintf

## Example

```
#include <stdio.h>

FILE *stream;
int i;
double fp;
char *s = "computer";
char c;
char buffer[200];

stream = fopen("results","w");
         .
         .
         .
/* Format and print various data.
*/

printf("%s\n",s);
printf("%c\n",c);
printf("%d\n",i);
printf("%f\n",fp);
```

## Summary

```
#include <stdio.h>

int putc(c, stream);      /* write a character to stream */
int c;                    /* character to be written */
FILE *stream;             /* pointer to file structure */

int putchar(c);           /* write a character to stdout */
int c;                    /* character to be written */
```

## Description

The **putc** routine writes the single character *c* to the output *stream* at the current position. **Putchar** is identical to **putc**(*c*, **stdout**).

## Return Value

**Putc** and **putchar** return the character written. A return value of **EOF** indicates an error. Since the **EOF** value is a legitimate integer value, the **ferror** function should be used to verify that an error occurred.

## See Also

**fputc, fputchar, getc, getchar**

---

*Note*

**Putc** and **putchar** are identical to **fputc** and **fputchar**, but are macros, not functions.

---

## Example

```
#include <stdio.h>

FILE *stream;
char buffer[81];
int i, ch;
          .
          .
          .
/* The following statements write a buffer to
** a stream.
*/

for (i = 0; (i < 81) && ((ch = putc(buffer[i],stream)) != EOF); i++
          ;

/* Note that the body of the for statement is null, since the
** write operation is carried out in the test expression.
*/
```

## Summary

```
#include <conio.h>          /* required only for function declarations */

void putch(c)
int c;                      /* character to be output */
```

## Description

The **putch** function writes the character *c* directly to the console.

## Return Value

There is no return value.

## See Also

**cprintf, getch, getche**

## Example

```
#include <conio.h>

/* This example shows how the getche function could be defined
** using putch and getch.
*/

int getche()
{
        int ch;

        ch = getch();
        putch(ch);
        return(ch);
}
```

# putenv

## Summary

```
#include <stdlib.h>          /* required only for function declarations */

int putenv(envstring);
char *envstring;             /* environment string definition */
```

## Description

The **putenv** function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program).

The *envstring* argument must be a pointer to a string with the form

*varname=string*

where *varname* is the name of the environment variable to be added or modified and *string* is the variable's value. If *varname* is already part of the environment, it is replaced by *string*; otherwise, the new *string* is added to the environment. A variable can be set to an empty value by specifying an empty *string*.

Do not free a pointer to an environment entry while the environment entry is still in use, or the environment variable will point into freed space. A similar problem can occur if you pass a pointer to a local variable to **putenv**, then exit the function in which the variable is declared.

## Return Value

**Putenv** returns zero if it is successful. A return value of −1 indicates an error.

## See Also

**getenv**

---

*Note*

The **getenv** and **putenv** functions use the global variable **environ** to access the environment table. **Putenv** may change the value of **environ**, thus invalidating the "envp" argument to the "main" function.

---

## Example

```
#include <stdlib.h>
#include <stdio.h>
#include <process.h>

/* Attempt to change an environment variable. */

if (putenv("PATH=a:\bin;b:\tmp") == -1) {
        printf("putenv failed -- out of memory");
        exit(1);
        }
```

# puts

## Summary

```
#include <stdio.h>

int puts(string);
char *string;          /* string to be output */
```

## Description

The **puts** function writes the given *string* to the standard output stream **stdout**, replacing the *string*'s terminating null character ('\0') with a newline character ('\n') in the output stream.

## Return Value

**Puts** returns the last character written, which is always the newline character ('\n'). A return value of **EOF** indicates an error.

## See Also

**fputs, gets**

## Example

```
#include <stdio.h>

int result;

/* The following statement writes a prompt to stdout. */

result = puts("insert data disk and strike any key");
```

## Summary

```
#include <stdio.h>

int putw(binint,stream);
int binint;                    /* binary int to be output */
FILE *stream;                  /* pointer to file structure */
```

## Description

The **putw** writes a binary value of type **int** to the current position of the specified *stream*. **Putw** does not affect the alignment of items in the stream, nor does it assume any special alignment.

## Return Value

**Putw** returns the value written. A return value of **EOF** may indicate an error. Since **EOF** is also a legitimate integer value, **ferror** should be used to verify an error.

## See Also

**getw**

---

*Note*

**Putw** is provided primarily for compatibility with previous libraries. Notice that portability problems may occur with **putw** since the size of an **int** and ordering of bytes within an **int** differ across systems.

---

## Example

```
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
          .
          .
          .
/* The following statement writes a word to a stream
** and checks for an error.
*/

putw(0345,stream);

if (ferror(stream)) {
          perror("putw failed");
          clearerr(stream);
          }
```

## Summary

```
#include <search.h>          /* required only for function declarations */

void qsort(base, num, width, compare);
char *base;
unsigned num, width;
int (*compare)( );
```

## Description

The **qsort** function implements a quicksort algorithm to sort an array of *num* elements, each of *width* bytes in size. *Base* is a pointer to the base of the array to be sorted. **Qsort** overwrites this array with the sorted elements.

*Compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **Qsort** will call the *compare* routine one or more times during the sort, passing pointers to two array elements on each call. The routine must compare the elements, then return one of the following values.

| Value | Meaning |
|---|---|
| Less than 0 | *element1* less than *element2* |
| 0 | *element1* equivalent to *element2* |
| Greater than 0 | *element1* greater than *element2* |

## Return Value

There is no return value.

## See Also

**bsearch**

## Example

```
#include <search.h>

int compare();   /* must declare as a function */

main (argc, argv)
int argc;
char **argv;
{
                    .
                    .
                    .

/* The following statement sorts the arguments in
** lexical order
*/
        qsort((char *)argv,argc,sizeof(char *),compare);
                    .
                    .
                    .
}

int compare (arg1, arg2)
char **arg1, **arg2;
{
        return(strcmp(*arg1,*arg2));
}
```

## Summary

```
#include <stdlib.h>   /* required only for function declarations */

int rand( );
```

## Description

The **rand** function returns a pseudo-random integer in the range 0 to $2^{15}-1$. The **srand** routine can be used before calling **rand** to set a random starting point.

## Return Value

**Rand** returns a pseudo-random number as described above. There is no error return.

## See Also

**srand**

## Example

```
#include <stdlib.h>
#include <stdio.h>

int x;

/* Print the first 20 random numbers generated.
*/

for (x = 1; x <= 20; x++)
        printf("iteration %d, rand=%d\n",x,rand());
```

# read

## Summary

```
#include <io.h>                    /* required only for function declarations */

int read(handle, buffer, count);
int handle;                        /* handle referring to open file */
char *buffer;                      /* storage location for data */
unsigned int count;                /* maximum number of bytes */
```

## Description

The **read** function attempts to read *count* bytes from the file associated with *handle* into *buffer*. The read operation begins at the current position of the file pointer (if any) associated with the given file. After the read operation, the file pointer points to the next unread character.

## Return Value

**Read** returns the number of bytes actually read, which may be less than *count* if there are fewer than *count* bytes left in the file or if the file was opened in text mode (see below). The return value 0 indicates an attempt to read at end-of-file. The return value −1 indicates an error, and **errno** is set to the following value.

| Value | Meaning |
|---|---|
| **EBADF** | The given *handle* is invalid; or the file is not open for reading; or the file is locked (MS-DOS Versions 3.0 or later only). |

If the file was opened in text mode, the return value may not correspond to the number of bytes actually read. When text mode is in effect, each carriage return/linefeed pair (CR-LF) is replaced with a single linefeed character (LF). Only the single linefeed character is counted in the return value. The replacement does not affect the file pointer.

## See Also

creat, fread, open, write

---

*Note*

Under MS-DOS, when files are opened in text mode, a CONTROL-Z character is treated as an end-of-file indicator. When the CONTROL-Z is encountered, the read terminates, and the next read returns 0 bytes. The file must be closed to clear the end-of-file indicator.

---

## Example

```
#include <io.h>
#include <stdio.h>

int fh, bytesread;
unsigned int nbytes = BUFSIZ;
char buffer[BUFSIZ];
          .
          .
          .
bytesread = read(fh,buffer,nbytes);
```

# realloc

## Summary

```
#include <malloc.h>          /* required only for function declarations */

char *realloc(ptr, size);
char *ptr;                   /* pointer to previously allocated  memory block */
unsigned size;               /* new size in bytes */
```

## Description

The **realloc** function changes the size of a previously allocated memory block. The *ptr* argument points to the beginning of the block. The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

The *ptr* argument may also point to a block that has been freed, as long as there has been no intervening call to **calloc**, **malloc**, or **realloc** since the block was freed.

## Return Value

**Realloc** returns a **char** pointer to the re-allocated memory block. The block may be moved when its size is changed; thus, the *ptr* argument to **realloc** is not necessarily the same as the return value.

The return value is **NULL** if there is insufficient memory available to expand the block to the given size. The original block is freed when this occurs.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than **char**, use a type cast on the return value.

## See Also

calloc, free, malloc

## Example

```
#include <malloc.h>
#include <stdio.h>

char *alloc;

/* Get enough space for 50 characters.
*/

alloc = malloc(50*sizeof(char));
          .
          .
          .
/* Reallocate block to hold 100 characters */

if (alloc != NULL)
        alloc = realloc(alloc,100*sizeof(char));
```

# rename

## Summary

```
#include <io.h>                    /* required only for function declarations */

int rename(newname, oldname);
char *newname;                     /* pointer to new name */
char *oldname;                     /* pointer to old name */
```

## Description

The **rename** function renames the file or directory specified by *oldname* to the name given by *newname*. *Oldname* must specify the pathname of an existing file or directory. *Newname* must not specify the name of an existing file or directory.

The **rename** function can be used to move a file from one directory to another by giving a different pathname in the *newname* argument. However, files cannot be moved from one device to another (for example, from Drive A to Drive B). Directories can only be renamed, not moved.

## Return Value

**Rename** returns 0 if it is successful. On an error, it returns a nonzero value and sets **errno** to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | File or directory specified by *newname* already exists or could not be created (invalid path); or *oldname* is a directory and *newname* specifies a different path. |
| ENOENT | File or pathname specified by *oldname* not found. |
| EXDEV | Attempt to move a file to a different device. |

## See Also

creat, fopen, open

## Example

```
#include <io.h>

int result;

/* The following statement changes the file "data" to
** have the name "input".
*/
result = rename("input","data");
```

# rewind

## Summary

```
#include <stdio.h>

int rewind(stream);
FILE *stream;          /* pointer to file structure */
```

## Description

The **rewind** function repositions the file pointer associated with *stream* to the beginning of the file. A call to **rewind** is equivalent to

```
fseek(stream, 0L, 0);
```

except that **rewind** clears the end-of-file and error indicators for the stream, and **fseek** does not.

## Return Value

**Rewind** returns the value 0 if the stream is successfully rewound. A non-zero return value indicates an error. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

## See Also

**fseek, ftell**

## Example

```
#include <stdio.h>

FILE *stream;
int data1, data2;
          .
          .
          .
/* Place data in the file */

fprintf(stream,"%d %d",data1,data2);

/* Now read the data file */

rewind(stream);
fscanf(stream,"%d",&data1);
```

# rmdir

## Summary

```
#include <direct.h>          /* required only for function declarations */

int rmdir(pathname);
char *pathname;              /* pathname of directory to be removed */
```

## Description

The **rmdir** function deletes the directory specified by *pathname*. The directory must be empty, and it must not be the current working directory or the root directory.

## Return Value

**Rmdir** returns the value 0 if the directory is successfully deleted. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | The given pathname is not a directory; or the directory is not empty; or the directory is the current working directory or root directory. |
| ENOENT | Pathname not found. |

## See Also

chdir, mkdir

## Example

```
#include <direct.h>

int result1, result2;

/* The following statements delete two directories:
** one at the root, and one in the current working
** directory.
*/

result1 = rmdir("/data");
result2 = rmdir("data");
```

## sbrk

### Summary

```
#include <malloc.h>  /* required only for function declarations */

char *sbrk(incr);
int incr;                    /* number of bytes added or subtracted */
```

### Description

The **sbrk** function resets the break value for the calling process. The break value is the address of the first byte of unallocated memory. **Sbrk** adds *incr* bytes to the break value; the size of the process's allocated memory is adjusted accordingly. Notice that *incr* may be negative, in which case the amount of allocated space is decreased by *incr* bytes.

### Return Value

**Sbrk** returns the old break value. A return value of −1 indicates an error, and **errno** is set to **ENOMEM**, indicating that insufficient memory was available.

### See Also

**calloc, free, malloc, realloc**

---

*Note*

In large model programs **sbrk** fails and returns −1. Use **malloc** for allocation requests in large model programs.

---

## Example

```
#include <malloc.h>
#include <stdio.h>

/* Allocate 100 bytes of memory.
*/

char *alloc;
alloc = sbrk(100);
        .
        .
        .
/* Now reduce allocated memory to 60 bytes.
*/

if (alloc != NULL)
        sbrk(-40);
```

# scanf

## Summary

```
#include <stdio.h>

int scanf(format-string [, argument...] );
char *format-string;                    /* format control */
```

## Description

The **scanf** function reads data from the standard input stream **stdin** into the locations given by *arguments*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields. *Format-string* may contain one or more of the following.

- Whitespace characters (blank (' '), tab ('\t'), or newline ('\n')). A whitespace character causes **scanf** to read, but not store, all consecutive whitespace characters in the input up to the next non-whitespace character. One whitespace character in the *format-string* matches any number (including zero) and combination of whitespace characters in the input.
- Non-whitespace characters, except for the percent sign character (%). A non-whitespace character causes **scanf** to read, but not store, a matching non-whitespace character. If the next character in **stdin** does not match, **scanf** terminates.
- Format specifications, introduced by the percent sign (%). A format specification causes **scanf** to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The *format-string* is read from left to right. Characters outside of format specifications are expected to match the sequence of characters in **stdin**; the matched characters in **stdin** are scanned but not stored. If a character in **stdin** conflicts with the *format-string,* **scanf** terminates. The conflicting character is left in **stdin** as if it had not been read.

When the first format specification is encountered, the value of the first input field is converted according to the format specification and stored in the location specified by the first *argument*. The second format specification causes the second input field to be converted and stored in the second *argument,* and so on through the end of the *format-string*.

An input field is defined as all characters up to the first whitespace character (space, tab, or newline), up to the first character that cannot be converted according to the format specification, or until the field *width,* if specified, is reached, whichever comes first. If there are too many arguments for the given format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the given format specifications.

A format specification has the following form.

%[*] [*width*] [*size-mod*] *type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, "%s").

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format control character, that character and following characters (up to the next percent sign) are treated as an ordinary sequence of characters — that is, a sequence of characters that must match the input. For example, to specify that a percent sign character is to be input, use "%%".

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *type.* The field is scanned but not stored.

*Width* is a positive decimal integer controlling the maximum number of characters to be read from **stdin.** No more than *width* characters are converted and stored at the corresponding *argument.* Fewer than *width* characters may be read if a whitespace character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before *width* is reached.

*Size-mod,* if present, is either "l", indicating that the **long** version of the following *type* is to be used, or "h", indicating that the **short** version is to be used. The corresponding *argument* should point to a **long** or **double** object (for the "l" character) or a **short** object (with the "h" character). The "l" and "h" modifiers may be used with the **d**, **o**, **x**, and **u** *type* characters. The "l" modifier may also be used with the **e** and **f** *type* characters. The "l" and "h" modifiers are ignored if specified for any other *type.*

The *type* characters and their meanings are as follows:

| Character | Type of Input Expected | Type of Argument |
|---|---|---|
| d | Decimal integer. | Pointer to **int** |
| D | Decimal integer. | Pointer to **long** |
| o | Octal integer. | Pointer to **int** |
| O | Octal integer. | Pointer to **long** |
| x | Hexadecimal integer. | Pointer to **int** |
| X | Hexadecimal integer. | Pointer to **long** |
| u | Unsigned decimal integer. | Pointer to **unsigned int** |
| U | Unsigned decimal integer. | Pointer to **unsigned long** |
| e<br>f | Floating-point value consisting of an optional sign (+ or −); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent ("e" or "E") followed by a possibly signed integer value. | Pointer to **float** |
| E<br>F | Same as **e** and **f**, above. | Pointer to **double** |
| c | Character; whitespace characters that are ordinarily skipped are read when **c** is specified; to read the next non-whitespace character, use "%1s". | Pointer to **char** |
| s | String. | Pointer to character array large enough for input field plus a terminating null character ('\0'), which is automatically appended |

To read strings not delimited by space characters, a set of characters in brackets ( [ ] ) may be substituted for the **s** (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that *does* appear in the rest of the character set.

To store a string without storing a terminating null character ('\0'), use the specification "%*n*c", where *n* is a decimal integer. In this case the **c** type character is taken to mean that the argument is a pointer to a character array. The next *n* characters are read from the input stream into the specified location, and no null character ('\0') is appended.

The **scanf** function scans each input field character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons — because the specified *width* has been reached, the next character cannot be converted as specified, the next character conflicts with a character in the control string that it is supposed to match, or the next character fails to appear (or appears) in a given character set. When this occurs, the next input field is considered to begin at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on **stdin**.

## Return Value

**Scanf** returns the number of fields that were successfully converted and assigned. The return value does not include fields which were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-file. A return value of 0 means that no fields were assigned.

## See Also

**fscanf, printf, sscanf**

## Example

```
#include <stdio.h>

FILE *stream;
int i;
float fp;
char s[81];
char c;

stream = fopen("data","r");
         .
         .
         .

/* Input various data.
*/

scanf("%s",s);
scanf(" %c",&c);
scanf("%d",&i);
scanf("%f",&fp);
```

## Summary

**#include <dos.h>**

**void segread**(*segregs*);
**struct SREGS** **segregs*;  /* segment register values */

## Description

The **segread** function fills the structure pointed to by *segregs* with the current contents of the segment registers. The function is intended to be used with the **intdosx** and **int86x** functions to retrieve segment register values for later use.

## Return Value

There is no return value.

## See Also

**intdosx, int86x, FP_SEG**

## Example

```
#include <dos.h>

struct SREGS segregs;
unsigned int cs, ds, es, ss;

/* The following statements get the current values of
** the segment registers.
*/

segread(&segregs);
cs = segregs.cs;
ds = segregs.ds;
es = segregs.es;
ss = segregs.ss;
```

# setbuf

## Summary

```
#include <stdio.h>

void setbuf(stream, buffer);
FILE *stream;                   /* pointer to file structure */
char *buffer;                   /* user-allocated buffer */
```

## Description

The **setbuf** function allows the user to control buffering for the specified *stream*. *Stream* must refer to an open file. If the *buffer* argument is **NULL**, the *stream* is unbuffered. If not, the *buffer* must point to a character array of length **BUFSIZ**, where **BUFSIZ** is the buffer size as defined in *stdio.h*. The user-specified *buffer* is used for input/output buffering instead of the default system-allocated buffer for the given *stream*.

The **stderr** and **stdaux** streams are unbuffered by default but may be assigned buffers with **setbuf**.

## Return Value

There is no return value.

## See Also

**fflush, fopen, fclose**

## Example

```
#include <stdio.h>

char buf[BUFSIZ];
FILE *stream1, *stream2;

stream1 = fopen("data1","r");
stream2 = fopen("data2","w");

/* Stream1 will use a user-assigned buffer, and
** stream2 will be unbuffered.
*/

setbuf(stream1,buf);
setbuf(stream2,NULL);
```

# setjmp

## Summary

```
#include <setjmp.h>

int setjmp(env);
jmp_buf env;          /* variable in which environment is stored */
```

## Description

The **setjmp** function saves a stack environment that can subsequently be restored using **longjmp**. **Setjmp** and **longjmp** provide a way to execute a nonlocal goto and are typically to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to **setjmp** causes the current stack environment to be saved in *env*. A subsequent call to **longjmp** restores the saved environment and returns control to the point just after the corresponding **setjmp** call. The values of all variables (except register variables) accessible to the routine receiving control contain the values they had when **longjmp** was called. The values of register variables are unpredictable.

## Return Value

**Setjmp** returns the value 0 after saving the stack environment. If **setjmp** returns as a result of a **longjmp** call, it returns the *value* argument of **longjmp**. There is no error return.

## See Also

**longjmp**

---

*Warning*

The values of register variables in the routine calling **setjmp** may not be restored to the proper values after a **longjmp** call is executed.

---

## Example

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

main()
{
        if (setjmp(mark) != 0) {
                printf("longjmp has been called\n");
                recover();
                exit(1);
                }
        printf("setjmp has been called\n");
                .
                .
                .
        p();
                .
                .
                .
}

p()
{
        int error = 0;
                .
                .
                .
        if (error != 0)
                longjmp(mark,-1);
                .
                .
                .
}

recover()
{
        /* ensure that data files won't be corrupted by
        ** exiting the program
        */
                .
                .
                .
}
```

# setmode

## Summary

```
#include <fcntl.h>
#include <io.h>                    /* required only for function declarations */

int setmode(handle, mode);
int handle;                        /* file handle */
int mode;                          /* new translation mode */
```

## Description

The **setmode** function sets the translation mode of the file given by *handle* to *mode*. The *mode* must be one of the following manifest constants.

| Manifest Constant | Meaning |
|---|---|
| O_TEXT | Set text (translated) mode. Carriage return/linefeed combinations (CR-LF) are translated into a single linefeed (LF) on input. Linefeed characters are translated into carriage return/linefeed combinations on output. |
| O_BINARY | Set binary (untranslated) mode. The above translations are suppressed. |

**Setmode** is typically used to modify the default translation mode of **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**, but can be used on on any file.

## Return Value

If successful, **setmode** returns the previous translation mode. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EBADF | Invalid file handle |
| EINVAL | Invalid *mode* argument (neither O_TEXT nor O_BINARY) |

## See Also

creat, fopen, open

## Example

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int result;

/* The following statement sets stdin to be binary
** (initially it is text).
*/

result = setmode(fileno(stdin),O_BINARY);
```

# signal

## Summary

```
#include <signal.h>

int (*signal(sig, func))( );
int sig;                        /* signal value */
int (*func)( );                 /* function to be executed */
```

## Description

The **signal** function allows a process to choose one of three ways to handle an interrupt signal from the operating system. The *sig* argument must be the manifest constant **SIGINT**, defined in *signal.h*. **SIGINT** corresponds to the MS-DOS interrupt signal, INT 23H (the CONTROL-C signal). The *func* argument must be one of the manifest constants **SIG_DFL** or **SIG_IGN** (defined in *signal.h*), or a function address. The action taken when the interrupt signal is received depends on the value of *func,* as follows.

| Value | Meaning |
|---|---|
| **SIG_IGN** | The interrupt signal is ignored. |
| **SIG_DFL** | The calling process is terminated and control returns to the MS-DOS command level. All files opened by the process are closed, but buffers are not flushed. |
| Function address | The function pointed to by *func* is passed the single argument **SIGINT** and executed. If the function returns, the calling process resumes execution just after the point where it received the interrupt signal. Before the specified function is executed, the value of *func* is set to **SIG_DFL**; the next interrupt signal is treated as described above for **SIG_DFL** unless an intervening call to **signal** specifies otherwise. This allows the user to reset signals in the called function if desired. |

## Return Value

**Signal** returns the previous value of *func*. A return value of −1 indicates an error, and **errno** is set to **EINVAL**, indicating an invalid *sig* value.

## See Also

**abort, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp**

---

*Note*

Signal settings are not preserved in child processes created by calls to **exec** or **spawn** routines. The signal settings are reset to the default in the child process.

---

## Example

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <process.h>

int handler();

main()
{
        if (signal(SIGINT,handler) == (int(*)())-1) {
                perror("couldn't set SIGINT");
                abort();
                }
                        .
                        .
                        .
}

int handler()
{
        char ch;

        signal(SIGINT,handler);
        printf("terminate processing? ");
        scanf("%1s",&ch);
        if (*ch == 'y' || *ch == 'Y')
                exit(0);
}
```

## Summary

```
#include <math.h>

double sin(x);          /* calculate sine of x */

double sinh(x);         /* calculate hyperbolic sine of x */

double x;               /* radians */
```

## Description

The **sin** and **sinh** functions return the sine and hyperbolic sine of $x$, respectively.

## Return Value

**Sin** returns the sine of $x$. If $x$ is large, a partial loss of significance in the result may occur. In such cases, **sin** generates a **PLOSS** error, but no message is printed. If $x$ is so large that a total loss of significance results, **sin** prints a **TLOSS** error message to **stderr** and returns 0. In both cases, **errno** is set to **ERANGE**.

**Sinh** returns the hyperbolic sine of $x$. If the result is too large, **sinh** sets **errno** to **ERANGE** and returns the value **HUGE** (positive or negative, depending on the value of $x$.)

Error handling can be modified by using the **matherr** routine.

## See Also

acos, asin, atan, atan2, cos, cosh, tan, tanh

## Example

```
#include <math.h>

double pi, x, y;

pi = 3.1415926535;
x = pi/2;
y = sin(x);     /* y is 1.0 */
          .
          .
          .
y = sinh(x);    /* y is 2.3 */
```

## Summary

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>                 /* required only for function declarations */

int sopen(pathname, oflag, shflag [, pmode]);
char *pathname;                 /* file pathname */
int oflag;                      /* type of operations allowed */
int shflag;                     /* type of sharing allowed */
int pmode;                      /* permission setting */
```

## Description

The **sopen** function opens the file specified by *pathname* and prepares the file for subsequent shared reading or writing as defined by *oflag* and *shflag*. *Oflag* is an integer expression formed by combining one or more of the following manifest constants, defined in *fcntl.h*. When more than one manifest constant is given, the constants are joined with the OR operator (|).

| Oflag | Meaning |
|---|---|
| O_APPEND | Reposition the file pointer to the end of the file before every write operation. |
| O_CREAT | Create and open a new file; this has no effect if the file specified by *pathname* exists. |
| O_EXCL | Return an error value if the file specified by *pathname* exists. Only applies when used with O_CREAT. |
| O_RDONLY | Open file for reading only; if this flag is given, neither O_RDWR nor O_WRONLY may be given. |
| O_RDWR | Open file for both reading and writing; if this flag is given, neither O_RDONLY nor O_WRONLY may be given. |
| O_TRUNC | Open and truncate an existing file to 0 length; the file must have write permission, and the contents of the file are destroyed. |
| O_WRONLY | Open file for writing only; if this flag is given, neither O_RDONLY nor O_RDWR may be given. |
| O_BINARY | Open file in binary (untranslated) mode. (See *fopen* for a description of binary mode.) |
| O_TEXT | Open file in text (translated) mode. (See *fopen* for a description of text mode.) |

---

*Note*

O_TRUNC destroys the complete contents of an existing file. Use with care.

---

*Shflag* is a constant expression consisting of one of the following manifest constants, defined in *share.h*. See your MS-DOS documentation for detailed information on sharing modes.

| Shflag | Meaning |
|---|---|
| SH_COMPAT | Set compatibility mode. |
| SH_DENYRW | Deny read and write access to file. |
| SH_DENYWR | Deny write access to file. |
| SH_DENYRD | Deny read access to file. |
| SH_DENYNONE | Permit read and write access. |

The *pmode* argument is required only when **O_CREAT** is specified. If the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. Otherwise, the *pmode* argument is ignored. The *pmode* argument is an integer expression containing one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in *sys\stat.h*. When both constants are given, they are joined with the OR operator (|). The meaning of the *pmode* argument is as follows.

| Value | Meaning |
|---|---|
| S_IWRITE | Writing permitted |
| S_IREAD | Reading permitted |
| S_IREAD \| S_IWRITE | Reading and writing permitted |

If write permission is not given, the file is read-only. Under MS-DOS all files are readable; it is not possible to give write-only permission. Thus, the modes **S_IWRITE** and **S_IREAD | S_IWRITE** are equivalent.

**Sopen** applies the current file permission mask to *pmode* before setting the permissions (see **umask**).

## Return Value

**Sopen** returns a file handle for the opened file. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Given pathname is a directory; or the file is read-only but an open for writing was attempted; or a sharing violation occurred (the file's sharing mode does not allow the specified operations; MS-DOS versions 3.0 or later only). |
| EEXIST | The O_CREAT and O_EXCL flags are specified but the named file already exists. |
| EINVAL | SHARE.COM not installed. |
| EMFILE | No more file handles available (too many open files). |
| ENOENT | File or pathname not found. |

## See Also

**close, creat, fopen, open, umask**

---

*Note*

The **sopen** function should be used only under MS-DOS version 3.0 or later. Under earlier versions of MS-DOS, the *shflag* argument is ignored.

File sharing modes will not work correctly for buffered files, so do not use **fdopen** to associate a file opened for sharing (or locking) with a stream.

---

## Example

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <share.h>
#include <io.h>

extern unsigned char _osmajor;
int fh;

        /* The _osmajor variable is used to test
        ** the MS-DOS version number before
        ** calling sopen.
        */

if (_osmajor >= 3)
        fh = sopen("data", O_RDWR | O_BINARY, SH_DENYRW);
else
        fh = open("data", O_RDWR | O_BINARY);
```

# spawnl – spawnvp

## Summary

```
#include <stdio.h>
#include <process.h>

int spawnl(modeflag, pathname, arg0, arg1...,argn, NULL);

int spawnle(modeflag, pathname, arg0, arg1...,argn, NULL, envp);

int spawnlp(modeflag, pathname, arg0, arg1...,argn, NULL);

int spawnv(modeflag, pathname, argv);

int spawnve(modeflag, pathname, argv, envp);

int spawnvp(modeflag, pathname, argv);

int modeflag;                    /* execution mode for parent process */
char *pathname;                  /* pathname of file to be executed */
char *arg0,*arg1,...,*argn;      /* list of pointers to arguments */
char *arv[ ];                    /* array of pointers to arguments */
char *envp[ ];                   /* array of pointers to environment settings */
```

## Description

The **spawn** functions create and execute a new child process. There must be enough memory available for loading and executing the child process. The *modeflag* argument determines the action taken by the parent process before and during the **spawn**. The following values for *modeflag* are defined in *process.h*.

| Value | Meaning |
|---|---|
| P_WAIT | Suspend parent process until execution of child process is complete |
| P_NOWAIT | Continue to execute parent process concurrently with child process |
| P_OVERLAY | Overlay parent process with child, destroying the parent (same effect as **exec** calls) |

Only the P_WAIT and P_OVERLAY *modeflag* values may currently be used. The P_NOWAIT value is reserved for possible future implementation. An error value is returned if P_NOWAIT is used.

The *pathname* argument specifies the file to be executed as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a filename. If *pathname* does not have a filename extension or end with a period (.), the **spawn** calls first append the extension ".COM" and search for the file; if unsuccessful, the extension ".EXE" is attempted. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the **spawn** calls search for *pathname* with no extension. The **spawnlp** and **spawnvp** routines search for *pathname* (using the same procedures) in the directories specified by the PATH environment variable.

Arguments are passed to the child process by giving one or more pointers to character strings as arguments in the **spawn** call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

The argument pointers may be passed as separate arguments (**spawnl, spawnle,** and **spawnlp**) or as an array of pointers (**spawnv, spawnve,** and **spawnvp**). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this argument is a copy of the *pathname* argument. (A different value will not produce an error.) Under versions of MS-DOS earlier than 3.0, the passed value of *arg0* or *arg*[0] is not available for use in the child process. However, under MS-DOS 3.0 and later, the *pathname* is available as *arg0* or *arg*[0].

The **spawnl, spawnle** and **spawnlp** calls are typically used in cases where the number of arguments is known in advance. *Arg0* is usually a pointer to *pathname*. *Arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn* there must be a **NULL** pointer to mark the end of the argument list.

**Spawnv, spawnve,** and **spawnvp** are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. *Argv*[0] is usually a pointer to the *pathname*. *Argv*[1] through *argv*[*n*] are pointers to the character strings forming the new argument list. *Argv*[*n*+1] must be a **NULL** pointer to mark the end of the argument list.

Files that are open when a **spawn** call is made remain open in the child process. In the **spawnl, spawnlp, spawnv,** and **spawnvp** calls, the child process inherits the environment of the parent. **Spawnle** and **spawnve** allow the user to alter the environment for the child process by passing a

list of environment settings through the *envp* argument. *Envp* is an array of character pointers, each element of which points to a null-terminated string defining an environment variable. Such a string has the form

NAME=*value*

where NAME is the name of an environment variable and *value* is the string value to which that variable is set. (Notice that *value* is not enclosed in double quotes.) When *envp* is **NULL**, the child process inherits the environment settings of the parent process.

## Return Value

The return value is the exit status of the child process. The exit status is 0 if the process terminated normally. The exit status can also be set to a nonzero value if the child process specifically calls the **exit** routine with a nonzero argument. If not set, a positive exit status indicates an abnormal exit via an **abort** or an interrupt.

A return value of −1 indicates an error (the child process is not started), and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| E2BIG | The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K bytes. |
| EINVAL | Invalid *modeflag* argument. |
| ENOENT | File or pathname not found. |
| ENOEXEC | The specified file is not executable or has an invalid executable file format. |
| ENOMEM | Not enough memory is available to execute the child process. |

*Note*

The **spawn** calls do not preserve the translation modes of open files. If the child process must use files inherited from the parent, the **setmode** routine should be used to set the translation mode of these files to the desired mode.

Signal settings are not preserved in child processes created by calls to **spawn** routines. The signal settings are reset to the default in the child process.

## See Also

abort, execl, execle, execlp, execv, execve, execvp, exit, _exit

## Example

```
#include <stdio.h>
#include <process.h>

extern char **environ;

char *args[4];
int result;

args[0] = "child";
args[1] = "one";
args[2] = "two";
args[3] = NULL;
          .
          .
          .
/* All of the following statements attempt to spawn a
** process called "child.exe" and pass it 3 arguments.
** The first 3 suspend the parent, and the last 3
** overlay the parent with the child.
*/

result = spawnl(P_WAIT,"child.exe","child","one","two",
         NULL);
result = spawnle(P_WAIT,"child.exe","child","one",
         "two",NULL,environ);
result = spawnlp(P_WAIT,"child.exe","child","one",
         "two",NULL);
result = spawnv(P_OVERLAY,"child.exe",args);
result = spawnve(P_OVERLAY,"child.exe",args,environ);
result = spawnvp(P_OVERLAY,"child.exe",args);
```

## Summary

```
#include <stdio.h>

int sprintf(buffer, format-string [, argument... ]);
char *buffer;                                   /* storage location for output */
char *format-string;                            /* format control string */
```

## Description

The **sprintf** function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in the *format-string*. The *format-string* consists of ordinary characters and has the same form and function as the *format-string* argument for the **printf** function; see the **printf** reference page for a description of the *format-string* and arguments.

## Return Value

**Sprintf** returns the number of characters printed.

## See Also

**fprintf, printf, sscanf**

## Example

```
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "computer";
char c;
          .
          .
          .
/* Format and print various data. */

j = sprintf(buffer, "%s\n",s);
j += sprintf(buffer+j, "%c\n",c);
j += sprintf(buffer+j, "%d\n",i);
j += sprintf(buffer+j, "%f\n",fp);
```

# sqrt

## Summary

```
#include <math.h>

double sqrt(x);
double x;                /* non-negative floating-point value */
```

## Description

The **sqrt** function calculates the square root of $x$.

## Return Value

**Sqrt** returns the square root result. If $x$ is negative, the function prints a **DOMAIN** error message to **stderr**, sets **errno** to **EDOM**, and returns 0.

Error handling can be modified by using the **matherr** routine.

## See Also

**exp, log, matherr, pow**

## Example

```
#include <math.h>
#include <stdlib.h>

double x, y, z;
          .
          .
          .
if ((z = sqrt(x+y)) == 0.0)
          if ((x+y) < 0.0)
                    perror("sqrt of a negative number");
```

## Summary

```
#include <stdlib.h>          /* required only for function declarations */

void srand(seed);
unsigned seed;               /* seed for random number generation */
```

## Description

The **srand** function sets the starting point for generating a series of pseudo-random integers. To re-initialize the generator, use 1 as the *seed* argument. Any other value for *seed* sets the generator to a random starting point.

The **rand** function is used to retrieve the pseudo-random numbers generated.

## Return Value

There is no return value.

## See Also

**rand**

## Example

```
#include <stdlib.h>
#include <stdio.h>

int x, ranvals[20];

/* Initialize the random number generator and save the
** first 20 random numbers generated in an array.
*/

srand(17);
for (x = 0; x < 20; ranvals[x++] = rand())
        ;
```

# sscanf

## Summary

```
#include <stdio.h>

int sscanf(buffer, format-string [, argument...] );
char *buffer;                                   /* stored data */
char *format-string;                            /* format control string */
```

## Description

The **sscanf** function reads data from *buffer* into the locations given by *arguments*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*. The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the **scanf** function; see the **scanf** reference page for a description of the *format-string*.

## Return Value

**Sscanf** returns the number of fields that were successfully converted and assigned. The return value does not include fields which were read but not assigned.

The return value is **EOF** for an attempt to read at end-of-string. A return value of 0 means that no fields were assigned.

## See Also

**fscanf, scanf, sprintf**

## Example

```
#include <stdio.h>

char *tokenstring = "15 12 14...";
int i;
float fp;
char s[81];
char c;
          .
          .
          .
/* Input various data.
*/

sscanf(tokenstring, "%s",s);
sscanf(tokenstring, " %c",&c);
sscanf(tokenstring, "%d",&i);
sscanf(tokenstring, "%f",&fp);
```

# stat

## Summary

```
#include <sys\types.h>
#include <sys\stat.h>

int stat(pathname, buffer);
char *pathname;                 /* pathname of existing file */
struct stat *buffer;            /* pointer to structure to receive results */
```

## Description

The **stat** function obtains information about the file or directory specified by *pathname* and stores it in the structure pointed to by *buffer*. The **stat** structure, defined in *sys\stat.h*, contains the following fields.

| Field | Value |
|---|---|
| **st_mode** | Bit mask for file mode information. **S_IFDIR** bit set if *pathname* specifies a directory. **S_IFREG** bit set if *pathname* specifies an ordinary file. User read/write bits set according to the file's permission mode. User execute bits are set using the filename extension. |
| **st_dev** | Drive number of the disk containing the file. |
| **st_rdev** | Drive number of the disk containing the file (same as **st_dev**). |
| **st_nlink** | Always 1. |
| **st_size** | Size of the file in bytes. |
| **st_atime** | Time of last modification of file. |
| **st_mtime** | Time of last modification of file (same as **st_atime**). |
| **st_ctime** | Time of last modification of file (same as **st_atime** and **st_mtime**). |

There are three additional fields in the **stat** structure type which do not contain meaningful values under MS-DOS.

## Return Value

**Stat** returns the value 0 if the file status information is obtained. A return value of −1 indicates an error, and **errno** is set to **ENOENT**, indicating that the filename or pathname could not be found.

## See Also

**access, fstat**

---

*Note*

If the given *pathname* refers to a device, the size and time fields in the **stat** structure are not meaningful.

---

## Example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

struct stat buf;
int result;
char *args[4];
          .
          .
          .
result = stat("child.exe",&buf);

if (result == 0)
        if (buf.st_mode == S_IEXEC)
                execv("child.exe", args);
```

## Summary

```
#include <string.h>                    /* required only for function declarations */

char *strcat(string1, string2);        /* append string2 to string1 */
char *string1;                         /* destination string */
char *string2;                         /* source string */

char *strchr(string, c);               /* search for first occurrence of c in string */
char *string;                          /* source string */
char c;                                /* character to be located */

int strcmp(string1, string2);          /* compare strings */
char *string1;
char *string2;

int strcmpi(string1, string2);         /* compare strings without regard to case */
char *string1;
char *string2;

char *strcpy(string1, string2);        /* copy string2 to string1 */
char *string1;                         /* destination string */
char *string2;                         /* source string */

int strcspn(string1, string2);         /* find first substring in string1
                                       ** of characters not in string2 */
char *string1;                         /* source string */
char *string2;                         /* character set */

char *strdup(string);                  /* duplicate string */
char *string;                          /* source string */
```

## Description

The **strcat, strchr, strcmp, strcmpi, strcpy, strcspn** and **strdup** functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string. No overflow checking is performed when strings are copied or appended.

**Strcat** appends *string2* to *string1,* terminates the resulting string with a null character, and returns a pointer to the concatenated string (*string1*).

**Strchr** returns a pointer to the first occurrence of *c* in *string.* The character *c* may be the null character ('\0'); the terminating null character of *string* is included in the search. The function returns **NULL** if the character is not found.

**Strcmp** compares *string1* and *string2* lexicographically and returns a value indicating their relationship, as follows.

| Value | Meaning |
|---|---|
| Less than 0 | *string1* less than *string2* |
| 0 | *string1* identical to *string2* |
| Greater than 0 | *string1* greater than *string2* |

**Strcmpi** is a case-insensitive version of **strcmp**. The two arguments *string1* and *string2* are compared without regard to case, meaning that the uppercase and lowercase forms of a letter are considered equivalent.

**Strcpy** copies *string2*, including the terminating null character, to the location specified by *string1*, and returns *string1*.

**Strcspn** returns the index of the first character in *string1* that belongs to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters not in *string2*. Terminating null characters are not considered in the search. If *string1* begins with a character from *string2*, **strcspn** returns 0.

**Strdup** allocates storage space (via a call to **malloc**) for a copy of *string* and returns a pointer to the storage space containing the copied string. The function returns **NULL** if storage could not be allocated.

## Return Value

The return values for these functions are described above.

## See Also

**strncat, strncmp, strncpy, strrchr, strspn**

## Example

```
#include <string.h>

char string[100], template[100], *result;
int numresult;
          .
          .
          .
/* Construct the string "computer program" using strcpy
** and strcat.
*/

strcpy(string,"computer");
result = strcat(string," program");

/* Search a string for the first occurence of 'a'.
*/

result = strchr(string,'a');

/* Determine whether a string is less than, greater
** than, or equal to another.
*/

numresult = strcmp(string,template);

/* Compare two strings without regard to case */

numresult = strcmpi("hello", "HELLO");   /* result is 0 */

/* Make a copy of a string.
*/

result = strcpy(template,string);

/* Search for  a's, b's, or c's in a string */

string = "xyzabbc";
result = strcspn(string,"abc");          /* result is 3 */

/* Make new string point to a duplicate of string.
*/

result = strdup(string);
```

## Summary

```
#include <string.h>          /* required only for function declarations */

int strlen(string);
char *string;                /* null-terminated string */
```

## Description

The **strlen** function returns the length in bytes of *string,* not including the terminating null character ('\0').

## Return Value

**Strlen** returns the *string* length. There is no error return.

## Example

```
#include <string.h>

char *string = "some space";
int result;
          .
          .
          .
/* Determine the length of a string.
*/

result = strlen(string); /* result = 10 */
```

## strlwr

### Summary

```
#include <string.h>   /* required only for function declarations */

char *strlwr(string);
char *string;         /* string to be converted */
```

### Description

The **strlwr** function converts any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

### Return Value

**Strlwr** returns a pointer to the converted *string*. There is no error return.

### See Also

**strupr**

### Example

```
#include <string.h>

char string[100], *copy;
          .
          .
          .
/* Make a copy of a string in lower case.
*/

copy = strlwr(strdup(string));
```

## Summary

```
#include <string.h>                    /* required only for function declarations */

char *strncat(string1, string2, n);    /* append n characters of string2 to string1 */
char *string1;                         /* destination string */
char *string2;                         /* source string */
unsigned int n;                        /* number of characters appended */

int strncmp(string1, string2, n );     /* compare first n character of strings */
char *string1;
char *string2;
unsigned int n;                        /* number of characters compared */

char *strncpy(string1, string2, n);    /* copy n characters of string2 to string1 */
char *string1;                         /* destination string */
char *string2;                         /* source string */
unsigned int n;                        /* number of characters copied */

char *strnset(string, c, n);           /* initialize first n characters of string */
char *string;                          /* string to be initialized */
char c;                                /* character setting */
unsigned int n;                        /* number of characters set */
```

## Description

The **strncat, strncmp, strncpy,** and **strnset** functions operate on at most the first *n* characters of null-terminated strings.

**Strncat** appends at most the first *n* characters of *string2* to *string1,* terminates the resulting string with a null character ('\0'), and returns a pointer to the concatenated string (*string1*). If *n* is greater than the length of *string2,* the length of *string2* is used in place of *n.*

**Strncmp** compares at most the first *n* characters of *string1* and *string2* lexicographically and returns a value indicating the relationship between the substrings, as listed below.

| Value | Meaning |
|---|---|
| Less than 0 | *substring1* less than *substring2* |
| 0 | *substring1* equivalent to *substring2* |
| Greater than 0 | *substring1* greater than *substring2* |

**Strncpy** copies exactly *n* characters of *string2* to *string1* and returns *string1*. If *n* is less than the length of *string2,* a null character ('\0') is *not* appended automatically to the copied string. If *n* is greater than the length of *string2,* the *string1* result is padded with null characters ('\0') up to length *n*.

**Strnset** sets at most the first *n* characters of *string* to the character *c* and returns a pointer to the altered *string*. If *n* is greater than the length of *string,* the length of *string* is used in place of *n*.

## See Also

**strcat, strcmp, strcpy, strset**

## Example

```
#include <string.h>

char copy[100], string[100], suffix[100], *result;
int numresult;
          .
          .
          .
/* Combine string with not more than 100 characters of
** suffix.
*/

result = strncat(string,suffix,100);

/* Determine the ordering of a string with respect to
** "program", but do not consider more than 7
** characters. So if string contains the prefix
** "program", strncmp will return zero. (Both
** "programmer" and "programming" will be 'equal' to
** "program".)
*/

numresult = strncmp(string,"program",7);

/* Copy at most 99 characters of a string.
*/

result = strncpy(copy,string,99);

/* Set the first 4 characters of a string to the
** character 'x'.
*/

result = strnset("computer",'x',4);

/* Result is now "xxxxuter".
*/
```

# strpbrk

## Summary

```
#include <string.h>                    /* required only for function declarations */

char *strpbrk(string1, string2);       /* find any character from string2 in string1 */
char *string1;                         /* source string */
char *string2;                         /* character set */
```

## Description

The **strpbrk** function finds the first occurrence in *string1* of any character from *string2*. The terminating null character ('\0') is not included in the search.

## Return Value

**Strpbrk** returns a pointer to the first occurrence of any character from *string2* in *string1*. A **NULL** pointer indicates that *string1* and *string2* have no characters in common.

## See Also

**strchr, strrchr**

## Example

```
#include <string.h>

char string[100], *result;
          .
          .
          .
/* Return a pointer to the first occurrence of either
** 'a' or 'b' in string.
*/

result = strpbrk(string,"ab");
```

## Summary

```
#include <string.h>          /* required only for function declarations */

char *strrchr(string, c);    /* find last occurrence of c in string */
char *string;                /* searched string */
char c;                      /* character to be located */
```

## Description

The **strrchr** function finds the last occurrence of the character *c* in *string*. The *string*'s terminating null character ('\0') is included in the search. (Use **strchr** to find the first occurrence of *c* in *string*.)

## Return Value

**Strrchr** returns a pointer to the last occurrence of *c* in *string*. A **NULL** pointer is returned if the given character is not found.

## See Also

**strchr, strpbrk**

## Example

```
#include <string.h>

char string[100], *result;
          .
          .
          .
/* Search a string for the last occurrence of 'a'.
*/

result = strrchr(string,'a');
```

## strrev

### Summary

```
#include <string.h>          /* required only for function declarations */

char *strrev(string);
char *string;                /* string to be reversed */
```

### Description

The **strrev** function reverses the order of the characters in the given *string*. The terminating null character ('\0') remains in place.

### Return Value

**Strrev** returns a pointer to the altered *string*. There is no error return.

### See Also

**strcpy, strset**

### Example

```
#include <string.h>

char string[100];
int result;
          .
          .
          .
/* Determine if a string is a palindrome (the same
** string read forwards and backwards).
*/

result = strcmp(string,strrev(strdup(string)));

/* If result==0 the string is a palindrome.
*/
```

## Summary

```
#include <string.h>          /* required only for function declarations */

char *strset(string, c);
char *string;                /* string to be set */
char c;                      /* character setting */
```

## Description

The **strset** function sets all characters of the given *string* except the terminating null character ('\0') to *c*.

## Return Value

**Strset** returns a pointer to the altered *string*. There is no error return.

## See Also

**strnset**

## Example

```
#include <string.h>
char string[100], *result;
          .
          .
          .
/* Set a string to be all blanks.
*/

result = strset(string,' ');
```

# strspn

## Summary

```
#include <string.h>          /* required only for function declarations */

int strspn(string1, string2);
char *string1;               /* searched string */
char *string2;               /* character set */
```

## Description

The **strspn** function returns the index of the first character in *string1* that *does not* belong to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters from *string2*. The null character ('\0') terminating *string2* is not considered in the matching process. If *string1* begins with a character not in *string2*, **strspn** returns 0.

## Return Value

**Strspn** returns an integer value specifying the position of the first character in *string1* not in *string2*.

## See Also

**strcspn**

## Example

```
#include <string.h>

char *string="cabbage";
int result;
          .
          .
          .
/* Determine the length of the prefix consisting of
** a's, b's, and c's.
*/

result = strspn(string,"abc");    /* result = 5 */
```

## Summary

```
#include <string.h>                    /* required only for function declarations */

char *strtok(string1, string2);        /* find token in string1 */
char *string1;                         /* string containing token(s) */
char *string2;                         /* set of delimiter characters */
```

## Description

The **strtok** function reads *string1* as a series of zero or more tokens and *string2* as the set of characters serving as delimiters of the tokens in *string1*. The tokens in *string1* may be separated by one or more of the delimiters from *string2*. The tokens are broken out of *string1* by a series of calls to **strtok**.

In the first call to **strtok** for a given *string1*, **strtok** searches for the first token in *string1*, skipping over leading delimiters. A pointer to the first token is returned.

To read the next token from *string1*, call **strtok** with a **NULL** value for the *string1* argument. The **NULL** *string1* argument causes **strtok** to search for the next token in the previous token string. The set of delimiters may vary from call to call, so *string2* can take any value.

## Return Value

The first time **strtok** is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, **strtok** returns a pointer to the next token in the string. A **NULL** pointer is returned when there are no more tokens. All tokens are null-terminated.

## See Also

**strcspn, strspn**

## Example

```
#include <string.h>
#include <stdio.h>

char *string="a  string,of ,,tokens ";
          .
          .
          .
/* The following loop gathers tokens (separated by
** blanks or commas) from a string until there are none
** left.
*/

token = strtok(string," ,");

while (token != NULL) {
          /* insert code to process the token here
          */
                    .
                    .
                    .
          token = strtok(NULL," ,"); /* get next token */
          }

/* Tokens returned are "a", "string", "of",
** and "tokens". The next call to strtok returns
** NULL and the loop terminates.
*/
```

## Summary

```
#include <string.h>          /* required only for function declarations */

char *strupr(string);
char *string;                /* string to be capitalized */
```

## Description

The **strupr** function converts any lowercase letters in the given *string* to uppercase. Other characters are not affected.

## Return Value

**Strupr** returns a pointer to the converted *string*. There is no error return.

## See Also

**strlwr**

## Example

```
#include <string.h>

char string[100], *copy;
          .
          .
          .
/* The following statement makes a copy of a string in
** uppercase.
*/

copy = strupr(strdup(string));
```

# swab

## Summary

```
#include <stdlib.h>                    /* required only for function declarations */

void swab(source, destination,n);
char *source;                          /* data to be copied and swapped */
char *destination;                     /* storage location for swapped data */
int n;                                 /* number of bytes copied */
```

## Description

The **swab** function copies *n* bytes from *source,* swaps each pair of adjacent bytes, and stores the result at *destination.* The integer *n* should be an even number to allow for swapping. **Swab** is typically used to prepare binary data for transfer to a machine that uses a different byte order.

## Return Value

There is no return value.

## See Also

**fgetc, fputc**

## Example

```
#include <stdlib.h>
#define NBYTES   1024

char from[NBYTES], to[NBYTES];

/* Copy n bytes from one location to another,
** swapping each pair of adjacent bytes.
*/

swab(from,to,NBYTES);
```

## Summary

```
#include <process.h>        /* required only for function declarations */

int system(string);
char *string;               /* command to be executed */
```

## Description

The **system** function passes the given *string* to the command interpreter (COMMAND.COM), which interprets and executes the string as an MS-DOS command. **System** refers to the COMSPEC and PATH environment variables to locate the MS-DOS file COMMAND.COM, which is used to execute the *string* command.

## Return Value

**System** returns the value 0 if *string* is successfully executed. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| E2BIG | The argument list for the command exceeds 128 bytes or the space required for the environment information exceeds 32K bytes. |
| ENOENT | COMMAND.COM cannot be found. |
| ENOEXEC | The COMMAND.COM file has an invalid format and is not executable. |
| ENOMEM | Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly. |

## See Also

execl, execle, execlp, execv, execve, execvp, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

## Example

```
#include <process.h>

int result;

/* The following statement appends a copy of the dos
** version number to a log file.
*/

result = system("ver >>result.log");
```

## Summary

```
#include <math.h>
double tan(x);          /* calculate tangent of x */
double tanh(x);         /* calculate hyperbolic tangent of x */
double x;               /* radians */
```

## Description

The **tan** and **tanh** functions return the tangent and hyperbolic tangent of *x*, respectively.

## Return Value

**Tan** returns the tangent of *x*. If *x* is large, a partial loss of significance in the result may occur. In such cases, **tan** sets **errno** to **ERANGE** and generates a **PLOSS** error, but no message is printed. If *x* is so large that a total loss of significance occurs, **tan** prints a **TLOSS** error message to **stderr**, sets **errno** to **ERANGE**, and returns 0.

**Tanh** returns the hyperbolic tangent of *x*. There is no error return.

## See Also

**acos, asin, atan, atan2, cos, cosh, sin, sinh**

## Example

```
#include <math.h>

double pi, x, y;

pi = 3.1415926535;
x = tan(pi/4.0);   /* x is 1.0 */

y = tanh(x);   /* y is 1.6 */
```

# tell

## Summary

```
#include <io.h>          /* required only for function declarations */

long tell(handle);
int handle;              /* handle referring to open file */
```

## Description

The **tell** function gets the current position of the file pointer (if any) associated with *handle*. The position is expressed as the number of bytes from the beginning of the file.

## Return Value

**Tell** returns the current position. A return value of −1L indicates an error, and **errno** is set to **EBADF** to indicate an invalid file handle argument. On devices incapable of seeking (such as terminals and printers), the return value is undefined.

## See Also

**ftell, lseek**

## Example

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>

int fh;
long position;

fh = open("data",O_RDONLY);
          .
          .
          .
position = tell(fh);      /* remember current position */
          .
          .
          .
lseek(fh, position, 0);  /* seek to previous position */
```

## Summary

```
#include <time.h>          /* required only for function declarations */

long time(timeptr);
long *timeptr;             /* storage location for time */
```

## Description

The **time** function returns the number of seconds elapsed since 00:00:00 Greenwich Mean Time, January 1, 1970, according to the system clock. The return value is also stored in the location given by *timeptr*. *Timeptr* may be **NULL**, in which case the return value is not stored.

## Return Value

**Time** returns the time in elapsed seconds. There is no error return.

## See Also

**asctime, ftime, gmtime, localtime, utime**

## Example

```
#include <time.h>
#include <stdio.h>

long ltime;

time(&ltime);
printf("the time is %s\n",ctime(&ltime));
```

## Summary

```
#include <ctype.h>
int toascii(c);          /* convert c to ASCII character */
int tolower(c);          /* convert c to lower case if appropriate */
int _tolower(c);         /* convert c to lower case */
int toupper(c);          /* convert c to upper case if appropriate */
int _toupper(c);         /* convert c to upper case */
int c;                   /* character to be converted */
```

## Description

The **toascii**, **tolower**, **_tolower**, **toupper**, and **_toupper** macros convert a single character as specified.

**Toascii** sets all but the low order 7 bits of *c* to 0 so that the converted value represents a character in the ASCII character set. If *c* already represents an ASCII character, *c* is unchanged.

**Tolower** converts *c* to lower case if *c* represents an uppercase letter. Otherwise, *c* is unchanged.

**_Tolower** is a version of **tolower** to be used only when *c* is known to be upper case. The result of **_tolower** is undefined if *c* is not an upper case letter.

**Toupper** converts *c* to upper case if *c* represents a lowercase letter. Otherwise, *c* is unchanged.

**_Toupper** is a version of **toupper** to be used only when *c* is known to be lower case. The result of **_toupper** is undefined if *c* is not a lowercase letter.

## Return Value

**Toascii, tolower, _tolower, toupper,** and **_toupper** return the possibly converted character *c*. There is no error return.

## See Also

**isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit**

---

*Note*

These routines are implemented as macros. However, **tolower** and **toupper** are also implemented as functions because the macro versions do not handle arguments with side effects correctly. The function versions can be used by removing the macro definitions through **#undef** directives or by not including *ctype.h*. Function declarations of **tolower** and **toupper** are given in *stdlib.h*.

---

## Example

```
#include <stdio.h>
#include <ctype.h>

int ch;

/* The following statements analyze all characters
** between code 0x0 and code 0x7f.  Toupper and tolower
** are applied to all codes. _Toupper and _tolower are
** applied to codes for which they make sense.
*/

for (ch = 0; ch <= 0x7f; ch++) {
        printf(" toupper=%#04x",toupper(ch));
        printf(" tolower=%#04x",tolower(ch));

        if (islower(ch))
                printf(" _toupper=%#04x",_toupper(ch));

        if (isupper(ch))
                printf(" _tolower=%#04x",_tolower(ch));

        putchar('\n');
        }
```

## Summary

```
#include <time.h>          /* required only for function declarations */

void tzset( );

int daylight;              /* daylight savings time flag */
long timezone;             /* difference in seconds from GMT */
char *tzname [2];          /* three-letter time zone strings */
```

## Description

The **tzset** function uses the current setting of the environment variable TZ to assign values to three variables, **daylight**, **timezone**, and **tzname**. These variables are used by the **ftime** and **localtime** functions to make corrections from Greenwich Mean Time (GMT) to local time.

The value of the environment variable TZ must be a three-letter time zone name, such as PST, followed by a possibly signed number giving the difference in hours between Greenwich Mean Time and local time. The number may be followed by a three-letter daylight savings time zone, such as PDT. For example, "PST8PDT" represents a valid TZ value for the Pacific time zone.

When **tzset** is called, the difference in seconds between Greenwich Mean Time and local time is stored in **timezone**. **daylight** is given a nonzero value if a daylight savings time zone is specified in the TZ setting; it is given the value 0 otherwise. **tzname** [0] is assigned the string value of the three-letter time zone name from the TZ setting; **tzname** [1] is assigned the string value of the daylight savings time zone. If the daylight savings time zone is omitted from the TZ setting, **tzname** [1] is assigned an empty string.

If TZ is not currently set, the default is "PST8PDT," which corresponds to the Pacific Time Zone. The default for **daylight** is 1; for **timezone** 28,800; for **tzname** [0] "PST"; and for **tzname** [1] "PDT".

## Return Value

There is no return value.

## See Also

**asctime, ftime, localtime**

## Example

```
int daylight;
long timezone;
char *tzname[];
          .
          .
          .
putenv("TZ=EST5");
tzset();                    /* daylight is 0,
                            ** timezone is 18000,
                            ** tzname [0] is "EST",
                            ** tzname [1] is empty
                            */
```

## Summary

```
#include <stdlib.h>                    /* required only for function declarations */

char *ultoa(value, string, radix);
unsigned long value;                   /* number to be converted */
char *string;                          /* string result */
int radix;                             /* base of value */
```

## Description

The **ultoa** function converts the digits of the given *value* to a null-terminated character string and stores the result in *string*. No overflow checking is performed. The *radix* argument specifies the base of *value;* it must be in the range 2–36.

## Return Value

**Ultoa** returns a pointer to *string*. There is no error return.

## See Also

**itoa, ltoa**

---

*Note*

The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes.

---

## Example

```
#include <stdlib.h>

int radix = 16;
char buffer[40];
char *p;

        /* p will be "501d9138 */
p = ultoa(1344115000L,buffer,radix);
```

# umask

## Summary

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>                    /* required only for function declarations */

int umask(pmode);
int pmode;                         /* default permission setting */
```

## Description

The **umask** function sets the file permission mask of the current process to the mode specified by *pmode*. The file permission mask is used to modify the permission setting of new files created by **creat**, **open** or **sopen**. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

*Pmode* is a constant expression containing one or both of the manifest constants **S_IWRITE** and **S_IREAD**, defined in *sys\stat.h*. When both constants are given, they are joined with the bitwise OR operator (|). The meaning of the *pmode* argument is as follows:

| Value | Meaning |
|---|---|
| S_IWRITE | Write permission |
| S_IREAD | Read permission |
| S_IREAD \| S_IWRITE | Read and write permission |

If the write bit is set in the mask, any new files will be read-only. Under MS-DOS all files are readable — it is not possible to give write-only permission. Thus, setting the read bit has no effect.

## Return Value

**Umask** returns the previous value of *pmode*. There is no error return.

## See Also

chmod, creat, mkdir, open

## Example

```
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>

int oldmask;

oldmask = umask(S_IWRITE);  /* create read-only files */
```

# ungetc

## Summary

```
#include <stdio.h>

int ungetc(c, stream);
int c;                          /* character to be pushed */
FILE *stream;                   /* pointer to file structure */
```

## Description

The **ungetc** function pushes the character *c* back onto the given input *stream*. The *stream* must be buffered and open for reading. A subsequent read operation on the *stream* starts with *c*. An attempt to push **EOF** on the stream using **ungetc** is ignored. **Ungetc** returns an error value if nothing has yet been read from *stream* or if *c* cannot be pushed back.

Characters placed on the stream by **ungetc** may be erased if an **fseek** or **rewind** function is called before the character is read from the *stream*.

## Return Value

**Ungetc** returns the character argument *c*. The return value **EOF** indicates a failure to push back the specified character.

## See Also

**getc, getchar, putc, putchar**

## Example

```
#include <stdio.h>
#include <ctype.h>

FILE *stream;
int ch;
int result = 0;
        .
        .
        .

/* The following statements gather a decimal integer
** from a stream.
*/

while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';

if (ch != EOF)
        ungetc(ch,stream); /* put non-digit back */
```

# ungetch

## Summary

```
#include <conio.h>    /* required only for function declarations */

int ungetch(c);
int c;                /* character to be pushed */
```

## Description

The **ungetch** function pushes the character *c* back to the console, causing *c* to be the next character read. **Ungetch** fails if it is called more than once before the next read.

## Return Value

**Ungetch** returns the character *c* if it is successful. A return value of **EOF** indicates an error.

## See Also

**cscanf, getch, getche**

## Example

```
#include <conio.h>
#include <ctype.h>

char buffer[100];
int count = 0;
int ch;

/* The following code gets a token, delimited by blanks
** newlines, from the keyboard.
*/

ch = getche();

while (isspace(ch))              /* skip preceding white space */
        ch = getche();

while (count < 99) {             /* gather token */
        if (isspace(ch))         /* end of token */
                break;

        buffer[count++] = ch;
        ch = getche();
        }

ungetch(ch);                     /* put back delimiter */
buffer[count] = '\0';            /* null terminate the token */
```

# unlink

## Summary

```
#include <io.h>                    /* required only for function declarations */

int unlink(pathname);
char *pathname;                    /* pathname of file to be removed */
```

## Description

The **unlink** function deletes the file specified by *pathname*.

## Return Value

**Unlink** returns the value 0 if the file is successfully deleted. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | Pathname specifies a directory or a read-only file. |
| ENOENT | File or pathname not found. |

## See Also

close

## Example

```
#include <io.h>
#include <stdlib.h>

int result;

result = unlink("tmpfile");
if (result == -1)
        perror("couldn't delete tmpfile");
```

## Summary

```
#include <sys\types.h>
#include <sys\utime.h>

int utime(pathname, times);
char *pathname;                 /* file pathname */
struct utimbuf *times;          /* pointer to stored time values */
```

## Description

The **utime** function sets the modification time for the file specified by *pathname*. The process must have write access to the file; otherwise, the time cannot be changed.

Although the **utimbuf** structure contains a field for access time, under MS-DOS, only the modification time is set. If *times* is a **NULL** pointer, the modification time is set to the current time. Otherwise, *times* must point to a structure of type **utimbuf**, defined in *sys\utime.h*. The modification time is set from the **modtime** field in this structure.

## Return Value

**Utime** returns the value 0 if the file modification time was changed. A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| **EACCES** | Pathname specifies directory or read-only file. |
| **EMFILE** | Too many open files (the file must be opened to change its modification time). |
| **ENOENT** | File or pathname not found. |

## See Also

asctime, ctime, fstat, ftime, gmtime, localtime, stat, time

## Example

```
#include <sys\utime.h>
#include <stdio.h>
#include <stdlib.h>

/* Set a file modification time to the current time.
*/

if (utime("/tmp/data",NULL) == -1)
        perror("utime failed");
```

## Summary

```
#include <io.h>                    /* required only for function declarations */

int write(handle, buffer, count);
int handle;                        /* handle referring to open file */
char *buffer;                      /* data to be written */
unsigned int count;                /* number of bytes */
```

## Description

The **write** function writes *count* bytes from *buffer* into the file associated with *handle*. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file is open for appending, the operation begins at the current end of the file. After the write operation, the file pointer (if any) is incremented by the number of bytes actually written.

## Return Value

**Write** returns the number of bytes actually written. The return value may be positive but less than *count* (for example, when running out of space on a disk before *count* bytes are written.) A return value of −1 indicates an error, and **errno** is set to one of the following values.

| Value | Meaning |
|---|---|
| EACCES | File is read-only or locked against writing |
| EBADF | Invalid file handle |
| ENOSPC | No space left on device |

If the given file was opened in text mode, each linefeed character (LF) is replaced with a carriage return/linefeed pair (CR-LF) in the output. The replacement does not affect the return value.

## See Also

**fwrite, open, read**

---

*Note*

When writing to files opened in text mode, a CONTROL-Z character is treated as the logical end of file. When writing to a device, a CONTROL-Z character in the buffer causes output to be terminated.

---

## Example

```
#include <io.h>
#include <stdio.h>

int fh, byteswritten;
unsigned int nbytes = BUFSIZ;
char buffer[BUFSIZ];
          .
          .
          .
byteswritten = write(fh,buffer,nbytes);
```

Library Reference

# Appendices

Appendices

# Appendices


A Error Messages 371

B A Common Library for XENIX and MS-DOS 377

# Appendix A
# Error Messages


A.1 Introduction 373
A.2 errno Values 374
A.3 Math Errors 376

# A.1 Introduction

This appendix lists and describes the values to which the **errno** variable can be set when an error occurs in a call to a library routine. Note that only some routines set the **errno** variable. The reference pages for the routines that set **errno** upon error explicitly mention the **errno** variable. (The reference pages are located in Part 2 of this manual.) If no mention of **errno** occurs, the routine does not set **errno**.

An error message is associated with each **errno** value. This message, along with a user-supplied message, can be printed by using the **perror** function.

The value of **errno** reflects the error value for the last call that set **errno**. The **errno** value is not automatically cleared by later successful calls. Thus, you should test for errors and print error messages, if desired, immediately after a call to obtain accurate results.

The include file *errno.h* contains the definitions of the **errno** values. However, not all of the definitions given in *errno.h* are used under MS-DOS. The full set of values is provided in the include file to maintain compatibility with the XENIX and UNIX include file having the same name.

This appendix lists only the **errno** values used under MS-DOS. For the complete listing of **errno** values, see the *errno.h* include file.

Also listed in this appendix are the errors produced by math routines when an error occurs. These errors correspond to the exception types defined in *math.h* and returned by the **matherr** function when a math error occurs.

# A.2 errno Values

The following list gives the **errno** values used on MS-DOS, the system error message corresponding to each value, and a brief description of the circumstances that cause the error.

| Value | Message | Description |
|---|---|---|
| E2BIG | Arg list too long. | The argument list exceeds 128 bytes or the space required for the environment information exceeds 32K bytes. |
| EACCES | Permission denied. | Access denied: the file's permission setting does not allow the specified access. This error can occur in a variety of circumstances; it signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file's attributes.<br>For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under MS-DOS 3.0 and later, **EACCES** may also indicate a locking or sharing violation.<br>The error can also occur in an attempt to rename a file or directory or to remove an existing directory. |
| EBADF | Bad file number. | The specified file handle is not a valid file handle value or does not refer to an open file; or an attempt was made to write to a file or device opened for read access (or vice versa). |

| | | |
|---|---|---|
| EDEADLOCK | Resource deadlock would occur. | Locking violation: the file cannot be locked after 10 attempts (MS-DOS Version 3.0 and later only). |
| EDOM | Math argument. | The argument to a math function is not in the domain of the function. |
| EEXIST | File exists. | The O_CREAT and O_EXCL flags are specified when opening a file, but the named file already exists. |
| EINVAL | Invalid argument. | An invalid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer is before the beginning of the file. |
| EMFILE | Too many open files. | No more file handles are available, so no more files can be opened. |
| ENOENT | No such file or directory. | The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a pathname does not specify an existing directory. |
| ENOEXEC | Exec format error. | An attempt is made to execute a file that is not executable or that has an invalid executable file format. |
| ENOMEM | Not enough core. | Not enough memory is available. This message can occur when insufficient memory is available to execute a child process or when the allocation request in an **sbrk** or **getcwd** call cannot be satisfied. |
| ENOSPC | No space left on device. | No more space for writing is available on the device (for example, the disk is full). |

| | | |
|---|---|---|
| ERANGE | Result too large. | An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the pathname argument to the **getcwd** function is longer than expected). |
| EXDEV | Cross-device link. | An attempt was made to move a file to a different device (using the **rename** function). |

## A.3 Math Errors

The following errors can be generated by the math routines of the C run-time library. These errors correspond to the exception types defined in *math.h* and returned by the **matherr** function when a math error occurs; see the **matherr** reference page in Part 2 of this manual for details.

| Error | Description |
|---|---|
| DOMAIN | An argument to the function is outside the domain of the function. |
| OVERFLOW | The result is too large to be represented in the function's return type. |
| PLOSS | A partial loss of significance occurred. |
| SING | Argument singularity: an argument to the function has an illegal value (for example, passing the value zero to a function that requires a nonzero value). |
| TLOSS | A total loss of significance occurred. |
| UNDERFLOW | The result is too small to be represented. |

# Appendix B
# A Common Library for XENIX and MS-DOS


B.1 Introduction 379

B.2 Common Run-Time Routines 379

B.2.1 Common Routines for MS-DOS and XENIX 379

B.2.2 Common Routines for MS-DOS and UNIX System V 380

B.2.3 Routines Specific to MS-DOS 381

B.3 Common Global Variables 381

B.3.1 Common Variables for MS-DOS and XENIX 382

B.3.2 Common Variables for MS-DOS and UNIX System V 382

B.3.3 Variables Specific to MS-DOS 382

B.4 Common Include Files 383

B.4.1 Common Include Files for MS-DOS and XENIX 383

B.4.2 Common Include Files for MS-DOS and UNIX System V 383

B.4.3 Include Files Specific to MS-DOS 384

B.5 Differences Between Common Routines 384
B.5.1 abort 384
B.5.2 access 384
B.5.3 chdir 385
B.5.4 chmod 385
B.5.5 creat 386
B.5.6 exec 386
B.5.7 fopen, freopen 387
B.5.8 fread 387
B.5.9 fseek 388
B.5.10 fstat 388
B.5.11 ftell 389
B.5.12 ftime 390
B.5.13 fwrite 390
B.5.14 getpid 390
B.5.15 locking 390
B.5.16 lseek 391
B.5.17 open 391
B.5.18 read 392
B.5.19 signal 392
B.5.20 stat 392
B.5.21 system 393
B.5.22 umask 394
B.5.23 unlink 394
B.5.24 utime 394
B.5.25 write 394

# B.1 Introduction

This appendix lists and describes routines from the Microsoft C Run-Time Library for MS-DOS that operate compatibly with C library routines on XENIX systems. The routines provide an identical interface to a set of operations useful on both XENIX and MS-DOS.

The XENIX and MS-DOS common library routines operate compatibly with UNIX library routines as well. In addition, the Microsoft C Run-Time Library for MS-DOS contains several routines that are compatible with UNIX System V routines but that are not currently implemented on XENIX.

With the exception of error returns, the math functions in the Microsoft C Run-Time Library for MS-DOS operate compatibly with the XENIX routines by the same names. Error returns for most math routines in the MS-DOS library have been upgraded for compatibility with UNIX System V math error handling.

# B.2 Common Run-Time Routines

The sections below list routines from the MS-DOS C library that are compatible with XENIX and UNIX System V routines. Routines specific to the MS-DOS environment are also listed.

## B.2.1 Common Routines for MS-DOS and XENIX

The following is a list of the common routines for MS-DOS and XENIX. The MS-DOS routines are compatible with the XENIX routines by the same names, except that routines marked by an asterisk (*) have a slightly different operation or meaning in the MS-DOS environment than they do under XENIX. These differences are fully described in later sections of this appendix. Math routines marked with a dagger (†) implement UNIX System V-style error returns on MS-DOS.

abort*
abs
access*
acos†
asctime
asin†
assert
atan†
atan2†
atof
atoi
atol
bessel†, ††
bsearch
cabs
calloc
ceil
chdir*
chmod*
chsize
clearerr
close
cos†
cosh†
creat*
ctime
dup
dup2
ecvt
execl*
execle*
execlp*
execv*
execve*
execvp*
exit
exp
fabs
fclose
fcvt
fdopen
feof
ferror
fflush
fgetc
fgets
fileno
floor
fmod
fopen*
fprintf
fputc
fputs
fread*
free
freopen*
frexp
fscanf
fseek*
fstat*
ftell*
ftime*
fwrite*
gcvt
getc
getchar
getcwd
getenv
getpid*
gets
getw
gmtime
hypot
isalnum
isalpha
isascii
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
ldexp†
localtime
locking*
log†
log10†
longjmp
lseek*
malloc
mktemp
modf
open*
perror
pow†
printf
putc
putchar
puts
putw
qsort
rand
read*
realloc
rewind
sbrk
scanf
setbuf
setjmp
signal*
sin†
sinh†
sprintf
sqrt†
srand
sscanf
stat*
strcat
strchr
strcmp
strcpy
strcspn
strdup
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strtok
swab
system*
tan†
tanh†
time
toascii
tolower
toupper
tzset
_tolower
_toupper
umask*
ungetc
unlink*
utime*
write*

## B.2.2 Common Routines for MS-DOS and UNIX System V

The XENIX-compatible routines listed in the previous section are also compatible with the routines by the same names in UNIX System V environments. In addition, the following MS-DOS routines are compatible with UNIX System V routines by the same name. These routines are not implemented on XENIX.

matherr
memccpy
memchr
memcmp
memcpy
memset
putenv

---

* Operates differently or has different meaning under MS-DOS than under XENIX

† Implements UNIX System V-style error returns

†† Does not correspond to a single function but to six functions named j0, j1, jn, y0, y1, and yn

Note that most of the math functions in the MS-DOS library implement error handling in the same manner as the UNIX System V routines by the same name. The math routines marked with a dagger (†) in the list of common routines for MS-DOS and XENIX (see Section B.2.1) implement System V-style error handling.

### B.2.3 Routines Specific to MS-DOS

The routines listed below are available only in the MS-DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these routines.

| | | | | |
|---|---|---|---|---|
| bdos | flushall | isatty | segread | strnset |
| cgets | FP_OFF | itoa | setmode | strrev |
| cprintf | FP_SEG | kbhit | sopen | strset |
| cputs | fputchar | labs | spawnl | strupr |
| cscanf | getch | ltoa | spawnle | tell |
| dosexterr | getche | mkdir | spawnlp | ultoa |
| eof | inp | movedata | spawnv | ungetch |
| _exit | int86 | outp | spawnve | |
| fcloseall | int86x | putch | spawnvp | |
| fgetchar | intdos | rename | strcmpi | |
| filelength | intdosx | rmdir | strlwr | |

## B.3 Common Global Variables

The sections below list global variables used in the MS-DOS C library that are also used in XENIX and UNIX environments. The variables specific to the MS-DOS environment are also listed.

## B.3.1 Common Variables for MS-DOS and XENIX

The following is a list of global variables used in the run-time library and available in both the MS-DOS and XENIX environments.

**daylight** **environ** **errno** **sys_errlist**
**sys_nerr** **timezone** **tzname**

---

*Note*

Not all values of **errno** available on XENIX are used by the MS-DOS run-time library.

---

## B.3.2 Common Variables for MS-DOS and UNIX System V

The XENIX-compatible global variables listed in the Section B.3.1 are also available in UNIX System V environments. There are no additional common variables for MS-DOS and UNIX System V.

## B.3.3 Variables Specific to MS-DOS

The following global variables are available only in the MS-DOS C library. Programmers who are writing code to be ported to XENIX systems should avoid using these variables.

**_doserrno**

**_fmode**

**_osmajor**

**_osminor**

**_psp**

# B.4 Common Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. Include files provided with the MS-DOS C library are compatible with include files by the same names on XENIX and UNIX systems. Some additional include files are compatible with include files by the same name in UNIX System V environments.

Sections B.4.1 and B.4.2 list the MS-DOS include files that are compatible with XENIX and UNIX System V. The include files that apply only to MS-DOS environments are listed in Section B.4.3.

## B.4.1 Common Include Files for MS-DOS and XENIX

The following MS-DOS include files are compatible with the XENIX (and UNIX) include files by the same name.

*assert.h*
*ctype.h*
*errno.h*
*fcntl.h*
*math.h*
*setjmp.h*
*signal.h*
*stdio.h*
*time.h*
*sys\locking.h*
*sys\stat.h*
*sys\timeb.h*
*sys\types.h*

## B.4.2 Common Include Files for MS-DOS and UNIX System V

The XENIX-compatible include files listed in Section B.4.1 are also compatible with the include files by the same names in UNIX System V environments. In addition, the names of the following MS-DOS include files correspond to UNIX System V include files; however, the MS-DOS include files may not contain all the constants and types defined in the corresponding UNIX System V include files.

*malloc.h*
*memory.h*
*search.h*
*string.h*

### B.4.3 Include Files Specific to MS-DOS

The following include files are used only in MS-DOS environments and do not have counterparts on XENIX and UNIX systems.

| | | |
|---|---|---|
| *conio.h* | *io.h* | *stdlib.h* |
| *direct.h* | *process.h* | *sys\utime.h* |
| *dos.h* | *share.h* | *v2tov3.h* |

## B.5 Differences Between Common Routines

The Sections B.5.1 through B.5.25 explain how the MS-DOS routines in the common library for XENIX and MS-DOS differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions of MS-DOS functions provided in the reference section (Part 2 of this manual) and with the descriptions of the XENIX routines in the appropriate XENIX manual.

### B.5.1 abort

The MS-DOS version of the **abort** routine terminates the process by a call to an exit routine rather than through a signal. Control is returned to the parent (calling) process with an exit status of 3 and the message

```
Abnormal program termination
```

is printed to standard error. No core dump occurs on MS-DOS.

### B.5.2 access

The **access** routine checks the access to a given file. Under MS-DOS, the real and effective user IDs are non-existent. The permission (access) setting can be any combination of the following values.

| Value | Meaning |
|---|---|
| 04 | Read |
| 02 | Write |
| 00 | Check for existence |

The "Execute" access mode (01) is not implemented.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS.

## B.5.3 chdir

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

## B.5.4 chmod

The **chmod** routine can set the "owner" access permissions for a given file, but all other permission settings are ignored. The mode argument can be any one of the constant-expressions shown in the left column below; the equivalent XENIX value is shown in the right column.

| Constant-Expression | Meaning | XENIX Value |
|---|---|---|
| S_IREAD | Read by owner | 0400 |
| S_IWRITE | Write by owner | 0200 |
| S_IREAD \| S_IWRITE | Read and write by owner | 0000 |

The **S_IREAD** and **S_IWRITE** constants are defined in the *sys\stat.h* include file. Note that the OR operator ( | ) is used to combine these constants to form read and write permission.

If write permission is not given, the file is treated as a read-only file. Giving write-only permission is allowed, but has no effect; under MS-DOS, all files are readable.

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

## B.5.5 creat

The **creat** routine creates a new file or prepares an existing file for writing. If the file is created, the access permissions are set as defined by the mode argument. Only "owner" permissions are allowed (see **chmod** above).

In case of error, only the **EACCES**, **EMFILE**, and **ENOENT** values may be returned for **errno** on MS-DOS.

Use of the **open** routine is preferred over **creat** when creating or opening files in both MS-DOS and XENIX environments.

## B.5.6 exec

The MS-DOS versions of the **execl**, **execle**, **execlp**, **execv**, **execve**, and **execvp** routines overlay the calling process, as in the XENIX environment. If there is not enough memory for the new process, the **exec** routine will fail and return to the calling process. Otherwise, the new process begins execution.

Under MS-DOS, the exec routines *do not*:

- Use the close-on-exec flag to determine open files for the new process.
- Disable profiling for the new process (profiling is not available under MS-DOS).
- Pass on signal settings to the child process. Under MS-DOS, all signals (including signals set to be ignored) are reset to the default in the child process.

The combined size of all arguments (including the program name) in an **exec** routine under MS-DOS must not exceed 128 bytes.

In case of error, the **E2BIG**, **EACCES**, **ENOENT**, **ENOEXEC**, and **ENOMEM** values may be returned for **errno** on MS-DOS. In addition, the **EMFILE** value may be used; under MS-DOS, the file must be opened to determine whether it is executable.

## B.5.7 fopen, freopen

The MS-DOS versions of the **fopen** and **freopen** routines open stream files just as they do in the XENIX environment. However, under MS-DOS the following additional values for the *type* string are available.

| Value | Meaning |
|---|---|
| t | Opens the file in text mode. Opening a file in this mode causes translation of carriage return/linefeed (CR-LF) character combinations into a single linefeed (LF) on input. Similarly, on output, linefeeds are translated into CR-LF combinations. |
| b | Opens the file in binary mode. This mode suppresses translation. |

See the MS-DOS reference pages (in Part 2 of this manual) for the **fopen** and **freopen** routines to obtain more information on the default mode setting.

The MS-DOS and XENIX versions of these routines also differ in their interpretation of append mode ("a" or "a+"). When append mode is specified in the MS-DOS version of **fopen** or **freopen**, the file pointer is repositioned to the end of the file before any write operation. Thus, all write operations take place at the end of the file.

In the XENIX versions, all write operations take place at the current position of the file pointer. In append mode, the file pointer is initially positioned at the end of the file, but if the file pointer is later repositioned, write operations take place at the new position rather than at the end of the file.

## B.5.8 fread

The MS-DOS **fread** routine uses the low-level **read** function to carry out read operations. If the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the the CR-LF pairs have been replaced. Thus, the return value may not always correspond to the actual number of bytes read. This is considered normal and has no implications for detecting the end of the file.

## B.5.9 fseek

The MS-DOS version of the **fseek** routine moves the file pointer to the given position, just as in the XENIX environment. However, for streams opened in text mode, **fseek** has limited use because carriage return-linefeed translations can cause **fseek** to produce unexpected results. The only **fseek** operations guaranteed to work on streams opened in text mode are: seeking with an offset of zero relative to any of the origin values, or seeking from the beginning of the file with an offset value returned from a call to **ftell**.

## B.5.10 fstat

MS-DOS does not make as much information available for file handles as it does for full pathnames; thus, the MS-DOS version of **fstat** returns less useful information than the **stat** routine. The MS-DOS **fstat** routine can detect device files, but it must not be used with directories.

The structure returned by **fstat** contains the following members.

| Member | Meaning |
|---|---|
| **st_mode** | User read and write bits reflect the file's permission setting. The **S_IFCHR** bit is set for a device; otherwise, the **S_IFREG** bit is set. |
| **st_ino** | Not used. |
| **st_dev** | Either drive number of the disk containing the file, or the file handle in the case of a device. |
| **st_rdev** | Either drive number of the disk containing the file, or the file handle in the case of a device. |
| **st_nlink** | Always 1. |
| **st_uid** | Not used. |
| **st_gid** | Not used. |
| **st_size** | Size of the file in bytes. |
| **st_atime** | Time of last modification of file. |
| **st_mtime** | Time of last modification of file (same as **st_atime**). |
| **st_ctime** | Time of last modification of file (same as **st_atime** and **st_mtime**). |

In case of error, only the **EBADF** value may be returned for **errno** on MS-DOS.

## B.5.11 ftell

The MS-DOS version of the **ftell** routine gets the current file pointer position, just as in the XENIX environment. However, for streams opened in text mode, the value returned by **ftell** may not reflect the physical byte offset, since text mode causes carriage return-linefeed translation. The **ftell** routine can be used in conjunction with the **fseek** routine to remember and return to file locations correctly.

## B.5.12 ftime

Unlike the system time on XENIX systems, the MS-DOS system time does not include the concept of a default time zone. Instead, **ftime** uses the value of an MS-DOS environment variable named TZ to determine the time zone. The user can set the default time zone by setting the TZ variable. If TZ is not explicitly set, the default time zone corresponds to the Pacific Time Zone. See the reference page for **tzset** in Part 2 of this manual for details on the TZ variable.

## B.5.13 fwrite

The MS-DOS **fwrite** routine uses the low-level **write** function to carry out write operations. If the file was opened in text mode, every linefeed (LF) character in the output is replaced by a carriage return-linefeed (CR-LF) pair before being written. This does not affect the return value.

## B.5.14 getpid

The **getpid** routine returns a process-unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process ID returned by **getpid** in the XENIX environment.

## B.5.15 locking

The MS-DOS and XENIX versions of the **locking** routine differ in several respects, as listed below.

1. Under MS-DOS, it is not possible to lock a file only against write access; locking a region of a file prevents both reading and writing in that region. This means that setting **LK_RLCK** in the **locking** call is equivalent to setting **LK_LOCK**, and setting **LK_NBRLCK** is equivalent to setting **LK_NBLCK**.

2. On MS-DOS, specifying **LK_LOCK** or **LK_RLCK** will *not* cause a program to wait until the specified region of a file is unlocked. Instead, up to ten attempts are made to lock the file (one attempt per second). If the lock is still unsuccessful after 10 seconds, the **locking** function returns an error value.

   On XENIX, if the first attempt at locking fails, the locking process "sleeps" (suspends execution) and periodically "wakes" to attempt the lock again. There is no limit on the number of attempts, and the process can continue indefinitely.

3. On MS-DOS, locking of overlapping regions of a file is not allowed.

4. On MS-DOS, if more than one region of a file is locked, only one region can be unlocked at a time, and the region must correspond to a region that was previously locked. You cannot unlock more than one region at a time, even if the regions are adjacent.

## B.5.16 lseek

In case of error, only the **EBADF** and **EINVAL** values may be returned for **errno** on MS-DOS.

## B.5.17 open

The **open** routine opens a file handle for a named file, just as in the XENIX environment. However, two additional *oflag* values (**O_BINARY** and **O_TEXT**) are available and the **O_NDELAY** and **O_SYNCW** values are not available.

The **O_BINARY** flag causes the file to be opened in binary mode, regardless of the default mode setting. Similarly, the **O_TEXT** flag causes the file to be opened in text mode.

In case of error, only the **EACCES**, **EEXIST**, **EMFILE**, and **ENOENT** values may be used for **errno** on MS-DOS.

## B.5.18 read

The MS-DOS version of the **read** routine reads characters from the file given by a file handle, just as in the XENIX environment. However, if the file has been opened in text mode, **read** replaces each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the the CR-LF pairs have been replaced. Thus, the return value may not always correspond with the actual number of bytes read. This is considered normal and has no implications for detecting an end-of-file condition.

In case of error, only the **EBADF** value may be used for **errno** on MS-DOS.

## B.5.19 signal

The MS-DOS version of the **signal** routine can only handle the **SIGINT** signal. In MS-DOS, **SIGINT** is defined to be INT 23H (the CONTROL-C signal).

On MS-DOS, child processes executed through the **exec** or **spawn** routines do not inherit the signal settings of the parent process. All signal settings (including signals set to be ignored) are reset to the default settings in the child process.

The MS-DOS version of **signal** uses only the **EINVAL** for **errno**.

## B.5.20 stat

The **stat** routine returns a structure defining the current status of the given file or directory. The structure members returned by **stat** have the following names and meanings on MS-DOS.

| Value | Meaning |
|---|---|
| **st_mode** | User read and write bits reflect the file's permission setting. The **S_IFDIR** bit is set for a device; otherwise, the **S_IFREG** bit is set. |
| **st_ino** | Not used. |
| **st_dev** | Drive number of the disk containing the file. |
| **st_rdev** | Drive number of the disk containing the file. |
| **st_nlink** | Always 1. |
| **st_uid** | Not used. |
| **st_gid** | Not used. |
| **st_size** | Size of the file in bytes. |
| **st_atime** | Time of last modification of file. |
| **st_mtime** | Time of last modification of file (same as **st_atime**). |
| **st_ctime** | Time of last modification of file (same as **st_atime** and **st_mtime**). |

In case of error, only the **ENOENT** value may be returned for **errno** on MS-DOS.

## B.5.21 system

The **system** routine passes the given string to the operating system for execution. For MS-DOS to execute this string, the full pathname of the directory containing COMMAND.COM must be assigned to the COMSPEC or PATH environment variable. The **system** call returns an error if COMMAND.COM cannot be found using these variables.

In case of error, only the **E2BIG**, **ENOENT**, **ENOEXEC** and **ENOMEM** values may be returned for **errno** on MS-DOS.

## B.5.22 umask

The **umask** routine can set a mask for "owner" read and write access permissions only. All other permissions are ignored. (See the discussion of the **access** routine above for details.)

## B.5.23 unlink

The MS-DOS version of the **unlink** routine always deletes the given file. Since MS-DOS does not implement multiple "links" to the same file, unlinking a file is the same as deleting it.

In case of error, only the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS.

## B.5.24 utime

The MS-DOS **utime** routine sets the file modification time only; MS-DOS does not maintain a separate access time.

In case of error, the **EACCES** and **ENOENT** values may be returned for **errno** on MS-DOS. In addition, the **EMFILE** value may be used; under MS-DOS, the file must be opened to set the modification time.

## B.5.25 write

The **write** routine writes a specified number of characters to the file named by the given file handle, just as in the XENIX environment. However, if the file has been opened in text mode, every linefeed (LF) character in the output is replaced by a carriage return-linefeed (CR-LF) pair before being written. This does not affect the return value.

In case of error, only the **EBADF** and **ENOSPC** values may be returned for **errno** on MS-DOS.

# Index

# Index


/ (forward slash), 21
\(backslash), 21

Abnormal program termination, 83
abort, 59, 83
  differences from XENIX version, 384
abs, 64, 84
  macro version, 80
Absolute value
  abs, 84
  cabs, 103
  fabs, 140
  labs, 227
access, 41, 85
  differences from XENIX version, 384
Access mode, 145, 164, 177
acos, 25, 54, 87
Allocation. *See* Memory allocation.
Appending, 145, 164, 177, 259, 313
Appending strings. *See* Concatenating strings.
Arc cosine, 87
Arc sine, 90
Arc tangent, 93
Argument singularity, 243
Argument type-checking, 6, 17
  with variable number of arguments, 18
Arguments
  macro, 15
  notational conventions, 8
  variable number, 18
asctime, 63, 88
asin, 25, 54, 90
assert, 64, 91
assert.h, 64
  contents, 70
Assertions, 91
Assigning buffers, 302
atan, 25, 54, 93
atan2, 25, 54, 93
atof, 25, 40, 95
atoi, 40, 95
atol, 40, 95

Backslash (\), 21
bdos, 57, 97
bessel, 25, 54, 99
Binary int
  reading, 204
  writing, 279
Binary mode, 22, 31, 165, 177, 259, 306, 313
Binary search, 101
BINMODE.OBJ, 23, 31
Brackets, 8
Break value, 294
bsearch, 61, 101
Buffer manipulation, 37
  memccpy, 246
  memchr, 247
  memcmp, 248
  memcpy, 250
  memset, 251
  movedata, 257
Buffering, 42, 47
Buffers
  assigning, 302
  comparing, 248
  copying, 246, 250, 257
    overlapping moves, 250
  flushing, 151, 161
  initializing, 251
  searching, 247
BUFSIZ constant, 45, 76
Byte order, 346
BYTEREGS type, 72

cabs, 25, 54, 103
calloc, 55, 104
Capital letters, 9

Carriage return-linefeed translation.
*See* Text mode.
Carry flag, 97, 210, 212, 215, 217
Case significance, 20
Categories, 37
ceil, 25, 54, 105
cgets, 52, 106
Character classification and conversion
isalnum, 219
isalpha, 219
isascii, 219
iscntrl, 222
isdigit, 222
isgraph, 222
islower, 222
isprint, 222
ispunct, 222
isspace, 222
isupper, 222
isxdigit, 222
toascii, 352
tolower, 352
toupper, 352
_tolower, 352
_toupper, 352
Character conversion. *See* Character classification and conversion.
Character device, 221
Characters
converting to ASCII, 352
converting to lowercase, 352
converting to uppercase, 352
reading, 153, 194, 284
reading from console, 196
with echo, 197
reading from port, 209
ungetting, 360, 362
writing, 170, 273, 367
writing to console, 275
writing to port, 263
chdir, 40, 108
differences from XENIX version, 385
Child process, 133, 318
signal settings, 133, 318
translation mode, 133, 318
chmod, 41, 109
differences from XENIX version, 385
chsize, 41, 111
Classifying characters, 219, 222
clearerr, 19, 43, 113
Clearing end-of-file, 113
Clearing errors, 113
close, 49, 114
Closing files, 51, 114
Closing streams, 48, 141
Common library, 379
global variables, 381
include files, 383
routines, 379, 380
differences, 384
math, 379
Comparing buffers, 248
Comparing strings, 330, 335
Compatibility mode, 313
Compatibility with XENIX and UNIX, 20, 379
complex type, 33, 73
Concatenating strings, 330, 335
conio.h, 52
contents, 70
Console and port I/O, 43, 52
cgets, 106
cprintf, 116
cputs, 118
cscanf, 121
getch, 196
getche, 197
inp, 209
kbhit, 226
outp, 263
putch, 275
ungetch, 362
CONTROL-Z, 285, 367
Conversions
characters to ASCII, 352
characters to lowercase, 352
characters to uppercase, 352
floating-point numbers to integers and fractions, 256
floating-point numbers to strings, 129, 143, 193
from previous versions, 80
integers to strings, 224
long integers to strings, 240, 357
strings to floating-point values, 95
strings to lowercase, 334

Conversions *(continued)*
strings to uppercase, 345
Copying buffers, 246, 250, 257
overlapping moves, 250
Copying strings, 330, 335
cos, 25, 54, 115
cosh, 25, 54, 115
Cosine, 115
cprintf., 52, 116
*See also* printf family.
cputs, 52, 118
CR-LF translation.
*See* Text mode.
creat, 49, 119
differences from XENIX version, 386
Creating directories, 252
Creating files, 119, 259, 313
cscanf, 52, 121
*See also* scanf family.
ctime, 63, 123
_ctype variable, 71
Ctype routines, 219, 222
ctype.h, 38
contents, 70
Current working directory
changing, 108
getting, 198

Data conversion, 40
*See also* Conversions.
atof, 95
atoi, 95
atol, 95
ecvt, 129
fcvt, 143
gcvt, 193
include files, 40
itoa, 224
ltoa, 240
ultoa, 357
Data items
reading, 173
writing, 191
Date routines. *See* Time routines.
daylight variable, 29, 355
De-allocating memory, 175
Declarations. *See* Function declarations.
Default translation mode, 22, 31, 259, 306, 313
changing, 23
in child process, 133, 318
Deleting directories, 292
Deleting files, 364
Delimiters for pathname components, 21
Differences from previous versions, 80
direct.h, 41
contents, 71
Directories
changing, 108
creating, 252
deleting, 292
getting current working directory, 198
renaming, 288
Directory control, 40
chdir, 108
chmod, 109
getcwd, 198
mkdir, 252
rmdir, 292
unlink, 364
Directory names. *See* Pathnames.
DOMAIN, 243, 376
dos.h, 57
contents, 71
_doserrno variable, 30
DOSERROR type, 33, 72, 125
dosexterr, 24, 57, 125
MS-DOS considerations, 24
Duplicating a file handle, 127
Duplicating strings. *See* Copying strings.
Dynamic allocation. *See* Memory allocation.

E2BIG, 375
EACCES, 375
EBADF, 375
Echoing characters, 197
ecvt, 40, 129
EDEADLOCK, 375
EDOM, 375
EEXIST, 375
EINVAL, 375

Ellipses, 9
EMFILE, 375
End-of-file condition, 20
End-of-file
  low-level I/O, 131
  stream I/O, 147
    clearing, 113, 290
ENOENT, 375
ENOEXEC, 375
ENOMEM, 375
ENOSPC, 375
environ variable, 32, 200, 276
Environment table, 32, 65, 200, 276
Environment variables, 9, 200, 276
eof, 20, 49, 131
EOF constant, 45, 76
ERANGE, 376
errno variable, 19, 20, 30, 51, 65, 72, 264, 373
  description of values used, 374
errno.h, 373
  contents, 72
Error handling, 19
  logic errors, 91
  low-level I/O, 20, 51
  math routines, 5, 19, 55, 243
  MS-DOS error codes, 31
  MS-DOS system calls, 125
  perror, 264
  stream I/O, 19, 49, 113, 149
Error indicator, 19, 49, 113, 149, 290
Error messages, 373
  math functions, 373, 376
  user-supplied, 264
Error returns, 19
Euclidean distance, 208
exception type, 33, 73, 243
EXDEV, 376
exec family, 24, 59, 133
  differences between exec routines, 60
  differences from XENIX versions, 386
  limitations on argument type-checking, 18
  pathname delimiters, 21
execl, 59, 133
  *See also* exec family.
execle, 59, 133
  *See also* exec family.
execlp, 59, 133
  *See also* exec family.
execlp, 59, 133
  *See also* exec family.
Executing MS-DOS commands from within programs, 347
Executing programs from within programs, 133, 318
execv, 59, 133
  *See also* exec family.
execve, 59, 133
  *See also* exec family.
execvp, 59, 133
  *See also* exec family.
execvp, 59, 133
  *See also* exec family.
_exit, 59, 137
exit, 59, 137
exp, 25, 54, 139
Exponential functions
  exp, 139
  frexp, 180
  ldexp, 228
  log, 234
  log10, 234
  pow, 266
  sqrt, 324

fabs, 25, 54, 140
Far pointers, 167
fclose, 43, 141
fcloseall, 43, 141
fcntl.h, 50
  contents, 72
fcvt, 40, 143
fdopen, 43, 145
feof, 20, 43, 147
ferror, 19, 43, 149
fflush, 43, 151
fgetc, 43, 153
fgetchar, 43, 153
fgets, 43, 155
File handles, 50
  duplicating, 127
  for stream, 158
  predefined, 50
File handling, 41

File handling *(continued)*
access, 85
chmod, 109
chsize, 111
filelength, 157
fstat, 185
isatty, 221
locking, 231
mktemp, 254
rename, 288
setmode, 306
stat, 328
umask, 358
unlink, 364
File permission mask. *See* Permission mask.
FILE pointer, 42, 45
File pointer, 48, 51
positioning, 183, 187, 238, 290, 350
File status information, 185, 328
FILE type, 33, 45, 77
filelength, 41, 157
Filename conventions, 9, 20
fileno, 43, 158
Files
changing size of, 111
closing, 51, 114
creating, 119, 259, 313
deleting, 364
length of, 157
locking, 231
modification time, 365
opening, 50, 119, 259, 313
positioning file pointer, 238, 350
reading characters, 284
renaming, 288
status information, 185, 328
temporary, 254
Floating point not loaded, 26
Floating-point numbers
converting from strings, 95
converting to strings, 129, 143, 193
Floating-point support, 25
floor, 43, 160
flushall, 43, 161
Flushing buffers, 47, 151, 161
fmod, 25, 54, 163
_fmode variable, 23, 31
fopen, 43, 164
changing default translation mode, 23
differences from XENIX version, 387
Formatted I/O, 116, 121, 168, 181, 267, 296, 323, 326
Forward slash (/), 21
fprintf, 43, 168
*See also* printf family.
fputc, 43, 170
fputchar, 43, 170
fputs, 43, 172, 278
FP_OFF, 57, 167
FP_SEG, 57, 167
fread, 43, 173
differences from XENIX version, 387
free, 55, 175
freopen, 43, 177
differences from XENIX version, 387
frexp, 25, 54, 180
fscanf, 43, 181
*See also* scanf family.
fseek, 43, 183
differences from XENIX version, 388
fstat, 41, 185
differences from XENIX version, 388
ftell, 43, 187
differences from XENIX version, 389
ftime, 63, 189
differences from XENIX version, 390
Function declarations, 17, 69
Functions, 14
fwrite, 43, 191
differences from XENIX version, 390

gcvt, 40, 193
getc, 43, 194
getch, 52, 196
getchar, 43, 194
getche, 52, 197
getcwd, 40, 198
getenv, 64, 200
getpid, 59, 202
differences from XENIX version, 390
gets, 43, 203
getw, 43, 204
Global variables, 29
daylight, 29, 355

Global variables *(continued)*
_doserrno, 30
environ, 32, 200, 276
errno, 30, 72, 264, 373
_fmode, 31
in common library, 381
_osmajor, 31
_osminor, 31
_psp, 32
sys_errlist, 30, 72, 264
sys_nerr, 30, 264
timezone, 29, 355
tzname, 355
gmtime, 63, 206
Goto (nonlocal), 65, 235, 304
Greenwich Mean Time, 206

Handle. *See* File handle.
HUGE, 74
Hyperbolic cosine, 115
Hyperbolic sine, 311
Hyperbolic tangent, 349
hypot, 25, 54, 208
Hypotenuse, 208

I/O
*See also* Low-level I/O.
*See also* Stream I/O.
buffered, 42
console and port, 43
low-level, 43
stream, 42, 43
Identifiers, 8
Include files
assert.h, 64, 70
conio.h, 52, 70
ctype.h, 38, 70
direct.h, 41, 71
dos.h, 71
errno.h, 72
fcntl.h, 72
function declarations, 69
in common library, 383
io.h, 41, 50, 73
malloc.h, 55, 73
math.h, 55, 73
memory.h, 37, 74
naming conventions, 6
notational conventions, 8
process.h, 60, 74
search.h, 61, 75
setjmp.h, 64, 75
share.h, 75
signal.h, 60, 75
stdio.h, 45, 76
stdlib.h, 64, 77
string.h, 62, 78
sys\locking.h, 78
sys\stat.h, 41, 78
sys\timeb.h, 64, 79
sys\types.h, 64, 79
sys\utime.h, 64, 79
time.h, 64, 79
v2tov3.h, 80
Initializing buffers, 251
Initializing strings, 335, 341
inp, 52, 209
Input and output. *See* I/O.
int86, 57, 210
int86x, 57, 212
intdos, 57, 215
intdosx, 57, 217
Integers
converting to strings, 224, 249, 357
Interrupt signal, 308
Interrupts. *See* MS-DOS interrupts.
_iob array, 77
io.h, 41, 50
contents, 73
isalnum, 38, 219
isalpha, 38, 219
isascii, 38, 219
isatty, 41, 221
iscntrl, 38, 222
isdigit, 38, 222
isgraph, 38, 222
islower, 38, 222
isprint, 38, 222
ispunct, 38, 222
isspace, 38, 222
isupper, 38, 222
isxdigit, 38, 222
Italics, 8

itoa, 40, 224, 357

j0. *See* bessel.
j1. *See* bessel.
jmp_buf type, 33
jn. *See* bessel.

kbhit, 52, 226
Key sequences, 9
Keystroke test, 226
Keywords, 10

labs, 64, 227
ldexp, 25, 54, 228
Length
  of files, 157
  of strings, 333
Lines
  reading, 155, 203
  writing, 278
LINT_ARGS, 6, 17, 69
Local time corrections, 29, 229, 355
  default, 30
localtime, 63, 229
locking, 24, 41, 231
  differences from XENIX version, 390
locking.h. *See* sys\locking.h.
log function, 25, 54, 234
log10 function, 25, 54, 234
Logarithmic functions, 234
Long integers, 240, 357
Long pointers, 167
longjmp, 64, 235
Low-level I/O, 43, 49
  close, 114
  creat, 119
  dup, 127
  dup2, 127
  eof, 131
  error handling, 20, 51
  lseek, 238
  open, 259
  read, 284
  sopen, 313
  tell, 350
Low-level I/O *(continued)*
  write, 367
lseek, 49, 238
  differences from XENIX version, 391
ltoa, 40, 240

Macros, 14
malloc, 55, 242
malloc.h, 55
  contents, 73
Manifest constants, 15
Mask. *See* Permission mask.
Math errors, 376
Math routines, 25, 54
  acos, 87
  asin, 90
  atan, 93
  atan2, 93
  bessel, 99
  cabs, 54, 103
  ceil, 105
  cos, 115
  cosh, 115
  error handling, 5, 19, 55
  exp, 139
  fabs, 140
  floor, 160
  fmod, 163
  frexp, 180
  hypot, 208
  ldexp, 228
  log, 234
  log10, 234
  matherr, 243
  modf, 256
  pow, 266
  sin, 311
  sinh, 311
  sqrt, 324
  tan, 349
  tanh, 349
math.h, 40, 55
  contents, 73
matherr, 19, 54, 243
max macro, 80
memccpy, 37, 246
memchr, 37, 247

memcmp, 37, 248
memcpy, 37, 250
Memory allocation, 55
  calloc, 104
  free, 175
  malloc, 242
  realloc, 286
  sbrk, 294
memory.h, 37
  contents, 74
memset, 37, 251
min macro, 80
Miscellaneous routines, 64
  abs, 84
  assert, 91
  getenv, 200
  labs, 227
  longjmp, 235
  perror, 264
  putenv, 276
  rand, 283
  setjmp, 304
  srand, 325
  swab, 346
mkdir, 40, 252
mktemp, 41, 254
modf, 25, 54, 256
Modification time, 365
movedata, 37, 257
Moving buffers. *See* Copying buffers.
MS-DOS commands, 347
MS-DOS considerations, 24, 31
MS-DOS error codes, 31
MS-DOS interface routines, 57
  bdos, 97
  dosexterr, 125
  FP_OFF, 167
  FP_SEG, 167
  int86, 210
  int86x, 212
  intdos, 215
  intdosx, 217
  segread, 301
MS-DOS interrupts, 210, 212
MS-DOS system calls
  error handling, 125
  invoking, 97, 215, 217
MS-DOS version number, 31
Names, 10
Naming files, 254
NDEBUG, 65, 70, 91
_NFILE constant, 76
Nonlocal goto, 65, 235, 304
Notational conventions, 8
NULL, 45, 76

oflag. *See* Open flag.
open, 49, 259
  changing default translation mode, 23
  differences from XENIX version, 391
  limitations on argument type-checking, 18
Open flag, 259, 313
Opening files, 50, 119, 259, 313
Opening streams, 45, 145, 164, 177
Optional arguments, 8
_osmajor variable, 25, 31
_osminor variable, 25, 31
outp, 52, 263
Output. *See* I/O.
OVERFLOW, 243, 376
Overlapping moves, 250
Overlay of parent process, 318
O_BINARY, 23, 31
O_TEXT, 23

Parent process, 133, 318
Pathnames
  case, 20
  delimiters, 20, 21
  notational conventions, 9
Permission mask, 358
Permission setting, 119, 259, 313
  changing, 109
  masking, 358
  testing, 85
perror, 19, 64, 264
PLOSS, 243, 376
Port I/O. *See* Console and port I/O.
Portability, 20
  *See also* Compatibility.
Positioning file pointer, 183, 187, 238, 290, 350
pow, 25, 54, 266

Predefined handles, 50
Predefined stream pointers, 46
printf, 43, 267
  *See also* printf family.
printf family
  floating-point support, 26
  limitations on argument type-checking, 18
Printing. *See* Write operations.
Process control, 59
  abort, 83
  exec family, 133
  execl, 133
  execle, 133
  execlp, 133
  execv, 133
  execve, 133
  execvp, 133
  exit, 137
  getpid, 202
  signal, 308
  spawn family, 318
  spawnl, 318
  spawnle, 318
  spawnlp, 318
  spawnv, 318
  spawnve, 318
  spawnvp, 318
  system, 347
  _exit, 137
Process ID, 202
process.h, 60
  contents, 74
Programming examples, 10
Pseudo-random integers, 283, 325
_psp variable, 32
PSP (Program Segment Prefix), 32
putc, 43, 273
putch, 52, 275
putchar, 43, 273
putenv, 64, 276
puts, 43, 278
putw, 43, 279

qsort, 61, 281
Quotation marks, 10

rand, 64, 283
Random access, 183, 187, 238, 290, 350
Random number generator, 283, 325
read, 49, 284
  differences from XENIX version, 392
  end-of-file condition, 20
Read access. *See* Permission setting.
Read operations
  binary int from stream, 204
  character from stdin, 153, 194
  character from stream, 153, 194
  characters from file, 284
  data items from stream, 173
  formatted, 121, 181, 296, 326
  from console, 106, 121, 196
    checking for keystroke, 226
    with echo, 197
  from port, 209
  line from stdin, 203
  line from stream, 155
realloc, 55, 286
Redirection, 46, 51, 177
Registers. *See* Segment registers.
REGS type, 33, 72
Remainder function, 163
rename, 41, 288
Reopening streams, 177
Reversing strings, 340
rewind, 43, 290
rmdir, 40, 292

sbrk, 55, 294
scanf, 43, 296
  *See also* scanf family.
scanf family
  floating-point support, 26
  limitations on argument type-checking, 18
Scanning. *See* Read operations.
search.h, 61
  contents, 75
Searching and sorting, 61
  bsearch, 101
  qsort, 281
Searching buffers, 247
Searching strings, 330, 338, 339, 342
Seed, 325

Segment registers, 301
segread, 57, 301
setbuf, 43, 302
setjmp, 64, 304
setjmp.h, 64, 75
setmode, 23, 41, 306
Setting buffers. *See* Assigning buffers.
Setting characters. *See* Initializing strings.
share.h, 75
Sharing flag, 313
Side effects, 14, 39
signal, 59, 308
  differences from XENIX version, 392
Signal settings, 133, 308, 318
signal.h, 60, 75
sin, 25, 54, 311
Sine, 311
SING, 243, 376
sinh, 25, 54, 311
sopen, 24, 49, 313
  limitations on argument type-checking, 18
Sorting. *See* Searching and sorting.
spawn family, 24, 59, 318
  differences between spawn routines, 60
  limitations on argument type-checking, 18
  pathname delimiters, 21
spawnl, 59, 318
  *See also* spawn family.
spawnle, 59, 318
  *See also* spawn family.
spawnlp, 59, 318
  *See also* spawn family.
spawnv, 59, 318
  *See also* spawn family.
spawnve, 59, 318
  *See also* spawn family.
spawnvp, 59, 318
  *See also* spawn family.
sprintf, 43, 323
  *See also* printf family.
sqrt, 25, 55, 324
srand, 64, 325
SREGS type, 72
sscanf, 43, 326
sscanf *(continued)*
  *See also* scanf family.
Stack environment
  restoring, 235
  saving, 304
Standard auxiliary. *See* stdaux.
Standard error. *See* stderr.
Standard input. *See* stdin.
Standard output. *See* stdout.
Standard print. *See* stdprn.
Standard types, 33
  complex, 73
  DOSERROR, 72, 125
  exception, 73, 243
  FILE, 77
  jmp_buf, 75
  REGS, 72
  SREGS, 72
  stat, 78, 185, 328
  timeb, 189
  tm, 79, 206
  utimbuf, 79, 365
stat, 41, 328
  differences from XENIX version, 392
stat type, 78, 185, 328
stat.h. *See* sys\stat.h.
stdaux, 46
  buffering, 47
  changing translation mode, 23, 306
  file handle, 50
stderr, 46
  buffering, 47
  changing translation mode, 23, 306
  file handle, 50
stdin, 46
  buffering, 47
  changing translation mode, 23, 306
  file handle, 50
stdio.h, 45
  contents, 76
stdlib.h, 39, 40, 64
  contents, 77
stdout, 46
  buffering, 47
  changing translation mode, 23, 306
  file handle, 50
stdprn, 46
  buffering, 47

stdprn *(continued)*
changing translation mode, 23, 306
file handle, 50
strcat, 62, 330
strchr, 62, 330
strcmp, 62, 330
strcmpi, 62, 330
strcpy, 62, 330
strcspn, 62, 330
strdup, 62, 330
Stream I/O, 42, 43
*See also* Console and port I/O.
buffering, 47
clearerr, 113
error handling, 49
fclose, 141
fcloseall, 141
fdopen, 145
feof, 147
ferror, 149
fflush, 151
fgetc, 153
fgetchar, 153
fgets, 155
fileno, 158
flushall, 161
fopen, 164
fprintf, 168
fputc, 170
fputchar, 170
fputs, 172
fread, 173
freopen, 177
fscanf, 181
fseek, 183
ftell, 187
fwrite, 191
getc, 194
getchar, 194
gets, 203
getw, 204
printf, 267
putc, 273
putchar, 273
puts, 278
putw, 279
rewind, 290
scanf, 296
Stream I/O *(continued)*
setbuf, 302
sprintf, 323
sscanf, 326
ungetc, 360
Stream pointer, 42
Streams
appending, 146, 164, 177
buffering, 302
clearing errors, 113
closing, 48, 141
file handles for, 158
formatted I/O, 168, 181, 267, 296, 323, 326
opening, 45, 145, 164
positioning file pointer, 183, 187, 290
reading binary int value, 204
reading characters, 153, 194
reading data items, 173
reading lines, 155, 203
reopening, 177
rewinding, 290
stdaux, 46
stderr, 46
stdin, 46
stdout, 46
stdprn, 46
translation mode, 165, 177
ungetting characters, 360
writing binary int value, 279
writing characters, 170, 273
writing data items, 191
writing lines, 278
writing strings, 172
String manipulation, 62
strcat, 330
strchr, 330
strcmp, 330
strcmpi, 330
strcpy, 330
strcspn, 330
strdup, 330
strlen, 333
strlwr, 334
strncat, 335
strncmp, 335
strncpy, 335
strnset, 335

String manipulation *(continued)*
strpbrk, 338
strrchr, 339
strrev, 340
strset, 341
strspn, 342
strtok, 343
strupr, 345
string.h, 62
contents, 78
Strings, 21
comparing, 330, 335
concatenating, 330, 335
converting to floating-point values, 95
converting to lowercase, 334
converting to uppercase, 345
copying, 330, 335
initializing, 335, 341
length of, 333
reading, 155, 203
from console, 106
reversing, 340
searching, 330, 338, 339, 342
writing, 172, 278
to console, 116
strlen, 62, 333
strlwr, 62, 334
strncat, 62, 335
strncmp, 62, 335
strncpy, 62, 335
strnset, 62, 335
strpbrk, 62, 338
strrchr, 62, 339
strrev, 62, 340
strset, 62, 341
strspn, 62, 342
strtok, 62, 343
strupr, 62, 345
Subdirectory conventions, 20
Suspension of parent process, 62, 318
swab, 62, 64, 346
Syntax conventions. *See* Notational conventions.
sys subdirectory, 21
system, 59, 347
differences from XENIX version, 393
pathname delimiters, 21
System calls. *See* MS-DOS system calls.
System time. *See* Time.
sys\locking.h, 78
sys\stat.h, 41
contents, 78
sys\timeb.h, 64
contents, 79
sys\types.h, 64
contents, 79
sys\utime.h, 64
contents, 79
sys_errlist variable, 30, 72, 264
sys_nerr variable, 30, 264
tan, 25, 55, 349
tanh, 25, 55, 349
tell, 49, 350
Temporary files, 254
Terminal capabilities, 221
Terminating a process, 83, 137
Testing characters, 219, 222
Testing for character device, 221
Testing for keystroke, 226
Text mode, 22, 31, 165, 177, 259, 306, 313
time, 63, 351
Time routines, 63
asctime, 88
ctime, 123
ftime, 189
gmtime, 206
localtime, 229
time, 351
tzset, 355
utime, 365
time.h, 64
contents, 79
Time
converting from long integer to string, 123
converting from long integer to structure, 206, 229
converting from structure to string, 88
correcting for local time, 229
obtaining system time, 189, 351
setting global time variables, 355
timeb type, 33, 189

timeb.h. *See* sys\timeb.h.
timezone variable, 29, 355
TLOSS, 243, 376
tm type, 33, 79, 206
toascii, 38, 352
Tokens, 343
_tolower, 38, 352
tolower, 38, 39, 352
_toupper, 38, 352
toupper, 38, 39, 352
Translation mode, *See* Text mode.
Trigonometric functions
  acos, 87
  asin, 90
  atan, 93
  atan2, 93
  cos, 115
  cosh, 115
  hypot, 208
  sin, 311
  sinh, 311
  tan, 349
  tanh, 349
Type-checking. *See* Argument type-checking.
types.h. *See* sys\types.h.
Types. *See* Standard types.
TZ environment variable, 29, 229, 355
  default value, 30
tzname variable, 29, 355
tzset, 63, 355

ultoa, 40, 357
umask, 41, 358
  differences from XENIX version, 394
UNDERFLOW, 243, 376
ungetc, 43, 360
ungetch, 52, 362
Ungetting characters, 360, 362
UNIX operating system, 379
unlink, 41, 364
  differences from XENIX version, 394
Update, 146, 164, 177
utimbuf type, 33, 79, 365
  standard types, 33
utime, 63, 365
  differences from XENIX version, 394

utime.h. *See* sys\time.h.

v2tov3.h, 80
Variables. *See* Global variables or Environment variables.
Version number, 31

Word. *See* Binary int.
WORDREGS type, 72
write, 49, 367
  differences from XENIX version, 394
Write access. *See* Permission setting.
Write operations
  binary int to stream, 279
  character to console, 275, 362
  character to stdout, 170, 273
  character to stream, 170, 273, 360
  characters file, 367
  data items from stream, 191
  formatted, 116, 168, 267, 323
  line to stream, 278
  string to stream, 172
  to console, 116, 118, 275
  to port, 263

XENIX operating system, 379

y0. *See* bessel.
y1. *See* bessel.
yn. *See* bessel.

# Manual Report Form

Please tell Microsoft what you think about these manuals. Your comments and suggestions help us improve our products. Mail this questionnaire to:

**MICROSOFT Corporation**
**10700 Northup Way, Building #4**
**Box 97200**
**Bellevue, WA 98009**
**Attn: User Education, Systems Languages**

---

Respond to any or all of the following questions. If you want to make additional comments, use the back of this page or a separate page.

**Product Name:** **Version Number:**

1. Did you find errors in the manuals? Please give the page number and description.

2. Which sections of the manuals do you consider most important? Do you have any suggestions for improving these sections?

3. Is it easy to find the information you need? Is anything missing?

4. Are the manuals clear and easy to read? Please name specific sections that you found confusing.

5. What type of user are you?
   _____ First-time programmer
   _____ First-time programmer in this language, but experienced in at least one other language
   _____ Experienced programmer

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

# Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

## Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

## Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

## Software Description

**Microsoft Product** _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

### Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

## Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

---

**Microsoft Use Only**

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

---

MICROSOFT®